

# GOOGLE SHEETS PROGRAMMING WITH GOOGLE APPS SCRIPT



*Your Guide  
To Building Spreadsheet Applications  
In The Cloud*

MICHAEL MAGUIRE

# **Google Sheets Programming With Google Apps Script (2015 Revision In Progress)**

Your Guide To Building Spreadsheet  
Applications In The Cloud

Michael Maguire

©2016

# Tweet This Book!

Please help Michael Maguire by spreading the word about this book on [Twitter!](#)

The suggested hashtag for this book is [#googlespreadsheetprogramming](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#googlespreadsheetprogramming>

# Contents

<b>Chapter 1: Introduction</b> . . . . .	<b>1</b>
1.1 Google Sheets . . . . .	1
1.2 Google Apps Script (GAS) . . . . .	2
1.3 JavaScript or Google Apps Script? . . . . .	3
1.4 Summary Of Topics Covered . . . . .	3
1.5 Software Requirements For This Book . . . . .	5
1.6 Intended Readership . . . . .	5
1.7 Book Code Available On GitHub . . . . .	6
1.8 My Blog On Google Spreadsheet Programming . . . . .	7
1.8 Guideline On Using This Book . . . . .	7
1.9 2015 Update Notes . . . . .	8
<b>Chapter 2: Getting Started</b> . . . . .	<b>9</b>
2.1 Introduction . . . . .	9
2.2 Google Apps Script Examples . . . . .	9
2.2 Executing Code – One Function At A Time . . . . .	11
2.3 Summary . . . . .	14
<b>Chapter 3: User-Defined Functions</b> . . . . .	<b>15</b>
3.1 Introduction . . . . .	15
3.2 Built-in Versus User-Defined Functions . . . . .	16
3.3 Why Write User-Defined Functions . . . . .	17
3.4 What User-Defined Functions Cannot Do . . . . .	18
3.5 Introducing JavaScript Functions . . . . .	21
3.6 User-Defined Functions Versus JavaScript Functions . . . . .	25
3.7 Using JSDoc To Document Functions . . . . .	26

## CONTENTS

3.8 Checking Input And Throwing Errors . . . . .	26
3.9 Encapsulating A Complex Calculation . . . . .	29
3.10 Numeric Calculations . . . . .	31
3.11 Date Functions . . . . .	33
3.12 Text Functions . . . . .	39
3.13 Using JavaScript Built-In Object Methods . . . . .	42
3.14 Using A Function Callback . . . . .	43
3.15 Extracting Useful Information About The Spreadsheet	45
3.16 Using Google Services . . . . .	49
3.18 Summary . . . . .	50
<b>Chapter 4: Spreadsheets and Sheets . . . . .</b>	<b>52</b>
4.1 A Note On Nomenclature . . . . .	52
4.2 Native And Host Objects . . . . .	53
4.3 A Note On Method Overloading In Google Apps Script	53
4.5 Object Hierarchies . . . . .	54
4.6 <i>SpreadsheetApp</i> . . . . .	57
4.7 The <i>Spreadsheet</i> Object . . . . .	58
4.8 The <i>Sheet</i> Object . . . . .	62
4.9 Practical Examples Using <i>Spreadsheet</i> And <i>Sheet</i> Objects . . . . .	63
4.10 Summary . . . . .	69
<b>Chapter 5: The <i>Range</i> Object . . . . .</b>	<b>71</b>
5.1 Introduction . . . . .	71
5.2 Range Objects Are Complex . . . . .	73
5.3 Creating A <i>Range</i> Object . . . . .	74
5.4 Getting And Setting Range Properties . . . . .	75
5.5 The <i>Range.offset()</i> Method . . . . .	78
5.6 The Sheet Data Range . . . . .	80
5.7 Transferring Values Between JavaScript Arrays And Ranges . . . . .	82
5.8 Named Ranges . . . . .	89
5.9 Practical Examples . . . . .	91
5.11 Concluding Remarks . . . . .	104

## CONTENTS

5.12 Summary . . . . .	105
<b>Chapter 6: MySQL And JDBC . . . . .</b>	<b>106</b>
6.1 Introduction . . . . .	106
6.2 What Is <i>JDBC</i> ? . . . . .	107
6.3 MySQL Preliminaries . . . . .	108
6.4 Connecting to a Cloud MySQL Database from the <i>mysql</i> Client . . . . .	109
6.5 An Overview of JDBC . . . . .	111
6.6 Note on Code Examples . . . . .	112
6.7 Connecting to the Database . . . . .	113
6.8 Create, Load, Query, Update and Delete a Database Table . . . . .	116
6.9 Prepared Statements . . . . .	126
6.10 Transactions . . . . .	129
6.11 Database Metadata . . . . .	131
6.12 A Practical GAS Example . . . . .	133
6.13 Summary . . . . .	143
<b>Chapter 7: User Interfaces - Menus and Forms . . . . .</b>	<b>144</b>
7.1 Introduction . . . . .	144
7.2 Adding A Menu . . . . .	145
7.3 Building Forms With <i>HtmlService</i> . . . . .	147
7.3.3 Defining Form Layout in CSS . . . . .	158
7.4 Transferring Data from Google Sheets To an HtmlSer- vice Web Application . . . . .	163
7.5 Create Professional-looking Forms the Easy Way - Use Bootstrap . . . . .	173
7.6 Summary . . . . .	177
<b>Chapter 8: Google Drive, Folders, Files, And Permissions</b>	<b>178</b>
8.1 Introduction . . . . .	178
8.2 List Google Drive File And Folder Names . . . . .	180
8.3 Creating And Removing Files And Folders . . . . .	183
8.4 Adding Files To And Removing Files From Folders . . . . .	186

## CONTENTS

8.5 File And Folder Permissions . . . . .	190
8.6 Practical Examples . . . . .	197
8.7 Summary . . . . .	210
<b>Chapter 9: Email and Calendars . . . . .</b>	<b>211</b>
9.1 Introduction . . . . .	211
9.2 Sending An Email Using <i>MailApp</i> . . . . .	212
9.3 Sending An Email With An Attachment Using <i>MailApp</i>	214
9.4 <i>GmailApp</i> . . . . .	216
9.5 Calendars . . . . .	229
9.6 Summary . . . . .	236
<b>Appendix A: Excel VBA And Google Apps Script Com-</b>	
<b>parison . . . . .</b>	<b>237</b>
Introduction . . . . .	237
Spreadsheets and Sheets . . . . .	238
Ranges . . . . .	249
<b>Appendix B: Final Notes . . . . .</b>	<b>268</b>
Additional Resources . . . . .	268
JSLint . . . . .	269
Getting Source code For This Book From Github . . . . .	270
Blog Updates . . . . .	270

# Chapter 1: Introduction

## 1.1 Google Sheets

Google Sheets is one of the core components of Google cloud applications. If you have a Gmail account, you can create and share your spreadsheets with others, even with those who do not have a Gmail account. Google Sheets offers a comprehensive set of standard spreadsheet features and functions similar to those found in other spreadsheet applications such as Microsoft Excel. In addition, it also supports some novel features such as the very versatile *QUERY* function and regular expression functions such *REGEXMATCH*.

What really distinguished Google Sheets from desktop spreadsheet applications like Excel is its cloud nature. The spreadsheet application runs in a browser and the spreadsheet files themselves are stored remotely. The spreadsheet files can be shared with others in read-only or read-edit modes making them ideal collaborative tools. Spreadsheets form just one part, albeit an important one, of the Google suite of products. Others are Google Documents, Gmail, calendars, forms, and so on and all of these products are inter-operable at least to some degree resulting in a very productive environment perfectly suited to collaborative work.

When I began using Google Sheets back in 2010 it was quite limited in terms of data volume, speed and functionality. It has undergone significant development since then and got a [major upgrade in March 2014](#)<sup>1</sup>. If your experience of Google Sheets was negatively influenced by experience with earlier versions, I encourage you to try it again, I think you will notice a big improvement. The

---

<sup>1</sup><https://support.google.com/docs/answer/3544847?hl=en>



old 400,000 cell limit per spreadsheet is gone and is now at least 2,000,000. It will comfortably deal with tens of thousands of rows which is, I believe, quite acceptable for any spreadsheet. Other spreadsheet applications such as Excel can handle a million plus rows but when data volumes grow to this size, it is advisable to switch to a database or a dedicated statistical application to handle such data sizes.

## 1.2 Google Apps Script (GAS)

The Google Sheets application also hosts a programming language called Google Apps Script (GAS) that is executed, not in the browser but remotely on the Google cloud. Google define Google Apps Script as follows:

*“Google Apps Script is a JavaScript cloud scripting language that provides easy ways to automate tasks across Google products and third party services.”*

If Google Sheets is so feature-rich, you might wonder why it needs to host a programming language. Here are few reasons why GAS is needed:

- Write user-defined functions for Google Sheets
- Write simple “macro” type applications
- Develop spreadsheet-based applications
- Integrate other Google products and services
- Develop Graphical User Interfaces (GUIs) that can be run as web applications
- Interact with cloud-based relational databases via Google *JDBC* Services.

GAS plays a similar role in Google Sheets to that played by Visual Basic for Applications (VBA) in Excel. Both are hosted by their

respective applications and both are used to extend functionality and integrate with other applications and services.

## 1.3 JavaScript or Google Apps Script?

The emphasis here is on using GAS to enhance and control Google Sheets. Other Google services are discussed in the context of how they can be used with Google Sheets. Since GAS is JavaScript (Google describe it as a sub-set of JavaScript 1.8), there will inevitably be discussion of JavaScript as a programming language. There is, therefore, some discussion of JavaScript topics as they relate to the code examples given.

Regarding terminology, when discussing a general JavaScript feature, the code may be referred to as “JavaScript” but when dealing with a Google App specific example, it may be referred to as “Google Apps Script” or “GAS”. The meaning of whichever term is used should be clear from the context. For example, the *Spreadsheet* object is central to Google Sheets programming. It is, however, provided by the hosting environment and is not part of JavaScript itself. This duality of the programming language and the objects provided by the hosting environment is similar to JavaScript running on the web client and the Document Object Model (DOM) entities that it manipulates.

## 1.4 Summary Of Topics Covered

This book aims to provide the reader with a solid knowledge of the GAS language both as it applies to Google Sheets and how it is used to allow Google Sheets to inter-operate with other Google products and services as well as with relational databases.

Chapter 2 introduces the GAS language and sets the scene for the chapters that follow. One of the most important applications

of the hosted spreadsheet language, be it VBA or GAS, is to allow users to write user-defined functions (also known as custom functions). These are covered in depth in chapter 3 and this chapter merits careful reading for any readers not familiar with JavaScript. Functions are central to JavaScript and, by extension, to GAS.

The *Spreadsheet*, *Sheet*, and *Range* objects are crucial for manipulating Google Sheets with GAS and these objects are covered in depth in chapters 4 and 5. All subsequent chapters assume that the reader is comfortable with these objects, their methods and their uses.

Having covered the basics of user-defined functions and the fundamental spreadsheet objects, chapter 6 explains how GAS can be used to work with a back-end relational database (MySQL). Spreadsheets are great tools but they have their limitations and, as applications increase both in complexity and in data volume, there comes a point where a more robust data storage solution is needed that that offered by spreadsheets.

In order to build spreadsheet *applications* some sort of Graphical User Interface (GUI) is usually required. Chapters 7 and 8 cover menus, alerts, prompts and user forms created using Google *Html Service*. Creating forms with *Html Service* offers the opportunity to develop web skills such as HTML, CSS and client-side JavaScript. For those not experienced in frontend development, this material offers a gentle introduction to a very important skill.

In addition to being an excellent collaborative tool, Google Sheets is part of a larger set of applications with which it can interact. GAS written in Google Sheets can be used to manipulate other Google products such as Google Drive, and Google Calendar and this aspect of GAS programming is covered in chapters 9 and 10.

For those coming to GAS from Excel VBA, appendix A will be of especial interest as it gives example code and explanations of how to perform common spreadsheet programming tasks in both languages. Appendix B gives an example of a quite complex spread-

sheet application written in GAS that brings together much of the material covered earlier. It describes how to build an application that takes spreadsheet data as input and uses it to generate SQL for table creation and row insertion. Appendix C discusses additional GAS and JavaScript resources that will be of interest to readers.

## **1.5 Software Requirements For This Book**

Not many! Modern cloud-based applications such as Google Sheets greatly reduce the technical barriers for new entrants. In the old days, you might have needed a specific operating system running some proprietary and often expensive software to get started. Not anymore! Anyone with a modern browser, an internet connection and a Gmail account running on Windows, Mac OS X or any version of Linux will be able to follow along with this book and run the code examples given. The code examples should run in any modern browser but I mainly use Chrome and Firefox and never use Internet Explorer (IE) although I expect that all the code will run fine in any post IE7 version.

## **1.6 Intended Readership**

This book is written for those who wish to learn how to programmatically manipulate Google Sheets using GAS. I began learning GAS for two reasons. Firstly, I was using Google Sheets as a collaborative tool in my work and, secondly, I had become aware of the increasing importance of JavaScript generally and was keen to learn it. Being able to use the JavaScript-based language that is GAS in spreadsheets appealed to me because I was already quite experienced in programming Excel using VBA so I felt that learning GAS to manipulate Google Sheets would offer a familiar

environment. I reasoned that there are many experienced Excel VBA programmers around who might feel similarly so that is why I wrote this book. There are of course now many people who use Google products who are familiar with JavaScript but who may not know much about spreadsheets. This book might also be of interest to this group of users. This book assumes programming knowledge in some programming language though not necessarily JavaScript.

## 1.7 Book Code Available On GitHub

The emphasis on this book is on practical code examples. Some of the code examples are short and only practical in the sense that they exemplify a GAS feature. GAS is a “moving target” in that new features are added and deprecated frequently thereby making it difficult to keep all code up-to-date. At the time of writing, all the examples worked as expected but please [email me](mailto:mick@javascript-spreadsheet-programming.com)<sup>2</sup> if something is broken or if you get a warning of something having been deprecated. To allow readers to follow along, I have tried to document the code extensively using [JSDoc](http://usejsdoc.org/)<sup>3</sup> and in-line code comments. To run the examples, you can copy and paste from the book to the GAS Script Editor. This will work but I recommend getting the code from GitHub. All the code examples in this book are available on a Github repository created specifically for this updated version of the book. The user name is **Rotifer** and the repository name is **GoogleSpreadsheetProgramming\_2015**. The full URL is [here](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015).<sup>4</sup> You can use the Git command line tool to check out the repository to your local machine or simply copy the examples directly from the GitHub repository.

---

<sup>2</sup>[mick@javascript-spreadsheet-programming.com](mailto:mick@javascript-spreadsheet-programming.com)

<sup>3</sup>[usejsdoc.org/](http://usejsdoc.org/)

<sup>4</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015)

## 1.8 My Blog On Google Spreadsheet Programming

I maintain a [blog](#)<sup>5</sup> that pre-dates this book. The blog began in late 2010 so some of the early examples use deprecated or unsupported features. That said, the blog inspired this book and is actively maintained so it can be viewed as a complementary resource to this book. It is worth checking for new entries from time to time because I use it to explore and discuss new spreadsheet and GAS features and I cover some more advanced JavaScript material there.

### 1.8 Guideline On Using This Book

I learn best from examples so this book is heavily example-driven. I first like to see something working and then examine it and learn from it. In order to derive maximum benefit from this book, it is important to execute the code from your own spreadsheets. Read the book descriptions and code documentation carefully and make sure that you understand both the objective of the code as well as how it works. Chapter 3 goes in to some depth on core JavaScript concepts such as functions, arrays and objects but if you are relatively inexperienced in JavaScript, then some background reading will help enormously. Remember, GAS is JavaScript so the better you understand JavaScript, the better you will be at writing and understanding GAS. Do not be afraid to experiment with code, change or re-write my examples at will! If you find better ways of doing things, please let me know!

---

<sup>5</sup><http://www.javascript-spreadsheet-programming.com>

## 1.9 2015 Update Notes

The first version of this book was released in November 2013 and the feedback I have received on it has been largely positive. However, I realise that some of the material is now out-of-date. For example, *UiApp* and *DocsList* have both been deprecated since that version of the book was released. I am also aware that the book could be improved by better explanations, better examples and generally better writing. I hope this version delivers a better product to its readers.

As I stated earlier, Google Sheets and GAS are “moving targets” that are subject to constant change. I intend to release a new version of this book yearly from now on so that the material remains current and the overall quality of the book improves. I will continue to blog about new features and topics that I have not covered so far and, if these look like a good fit and attract interest from readers, I will incorporate such subjects into later version of the book.

Something I find annoying and expensive is when I buy a technical book only to learn a few months later that a new edition has been released. I have one shell programming book from 1990 that I still use but shell programming is one of the few stable technologies that I use (SQL is another, yes, features are added but the core is quite stable). Most technical books that I have bought in the past become obsolete, at least in part, in a year or two. My idea with this book is to make sure those who buy it once get all the updates for free and I plan to keep updating it indefinitely. Leanpub make this possible, so a big thanks to them and thanks also to all of you who bought the first version of this book!

Time to write some GAS!

# Chapter 2: Getting Started

## 2.1 Introduction

The best way to learn JavaScript/Google Apps Script is to write some code. Getting started is very straightforward: All that is needed is a Gmail account and a browser with an Internet connection. To run some example code, first go to Google Drive and create a spreadsheet. To view the script editor, select *Tools->Script editor...* from the spreadsheet menu bar. The first time you do this in a new spreadsheet file, you will be presented with a pop-up window entitled “Google Apps Script”, just ignore and close it for now. Give the project a name, any name you like, by hovering over and replacing the text “untitled project” on the top left. Delete the code stub entitled “myFunction” so that the script editor is now blank. Paste in the example code in the following sections and save (save icon or menu action *File->Save*).

## 2.2 Google Apps Script Examples

Here are four example functions. When pasted into the script editor, the code formatting applied by the editor becomes evident and makes the code easier to read. The code for this chapter can be viewed and downloaded from GitHub [here](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch02.gs).<sup>6</sup>

---

<sup>6</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015/blob/master/ch02.gs](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch02.gs)



```
1  function sayHelloAlert() {
2    // Declare a string literal variable.
3    var greeting = 'Hello world!',
4        ui = SpreadsheetApp.getUi();
5    // Display a message dialog with the greeting
6    //(visible from the containing spreadsheet).
7    // Older versions of Sheets used Browser.msgBox()
8    ui.alert(greeting);
9  }
10
11 function helloDocument() {
12   var greeting = 'Hello world!';
13   // Create DocumentApp instance.
14   var doc =
15     DocumentApp.create('test_DocumentApp');
16   // Write the greeting to a Google document.
17   doc.setText(greeting);
18   // Close the newly created document
19   doc.saveAndClose();
20 }
21
22 function helloLogger() {
23   var greeting = 'Hello world!';
24   //Write the greeting to a logging window.
25   // This is visible from the script editor
26   // window menu "View->Logs...".
27   Logger.log(greeting);
28 }
29
30
31 function helloSpreadsheet() {
32   var greeting = 'Hello world!',
33       sheet = SpreadsheetApp.getActiveSheet();
34   // Post the greeting variable value to cell A1
35   // of the active sheet in the containing
```

```
36 // spreadsheet.
37 sheet.getRange('A1').setValue(greeting);
38 // Using the LanguageApp write the
39 // greeting to cell:
40 // A2 in Spanish,
41 // cell A3 in German,
42 // and cell A4 in French.
43 sheet.getRange('A2')
44     .setValue(LanguageApp.translate(
45         greeting, 'en', 'es'));
46 sheet.getRange('A3')
47     .setValue(LanguageApp.translate(
48         greeting, 'en', 'de'));
49 sheet.getRange('A4')
50     .setValue(LanguageApp.translate(
51         greeting, 'en', 'fr'));
52 }
```

## 2.2 Executing Code – One Function At A Time

In order to execute code, there must be at least one valid function in the script editor. After pasting the code above, there are four functions that will each be executed in turn.

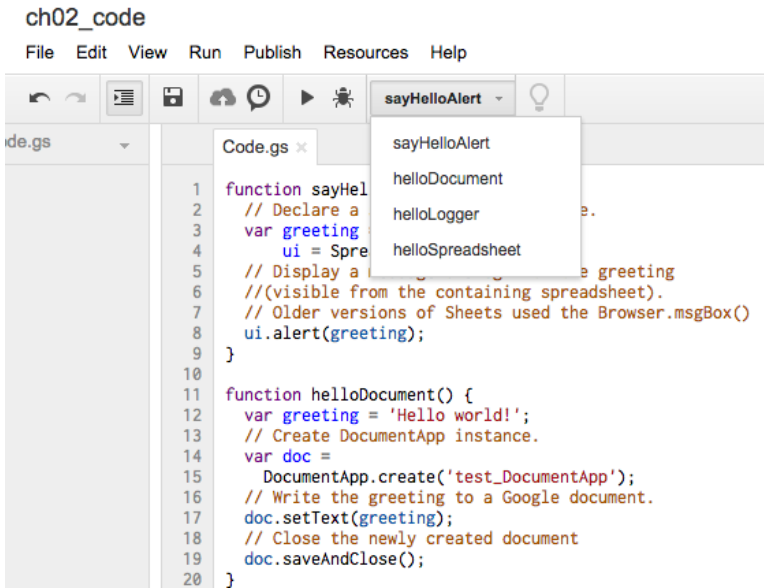


Figure 2-1: Google Apps Script Editor displaying the code and the “Select function” drop-down list.

Select function `sayHelloAlert()` from the “Select function” drop-down list on the script editor toolbar and then press the execute icon (to the left of the function list drop-down). You will need to authorize the script when you first try to execute it. Subsequent executions do not require authorisation. Once authorised, switch to the spreadsheet and you will see a small window with the greeting “Hello world”. These browser popup displays are modal meaning that they block all subsequent code execution until they are closed. For this reason, their use should be limited. The `Logger` is generally a better tool for writing and displaying output.



## New Sheets Feature

`Browser.msgBox` instead of `alert()` is used in older versions and still works.

Now select the second function named *helloDocument()* and execute it. This is a much more interesting example than the previous one because it shows how GAS written in one application can be used to manipulate other applications. The first time you try to execute it, you will get a message saying, “Authorization required”. Once you authorise it and then execute, it will create a new Google *Document* and write the message to it. This example, though trivial and useless, does demonstrate how GAS code written in one application can manipulate other applications. This is a very powerful feature and will be a recurring theme of this book.

The *helloLogger()* function, when executed, writes the message to a logging area that is viewable from the script editor menu “View->Logs...”. It is equivalent to the “console.log” in Firebug and Node.js. It will be used frequently in later code examples for output and for error reporting.

The final function *helloSpreadsheet()* demonstrates two important aspects of Google Apps Script:

Firstly, spreadsheets can be manipulated via the *Spreadsheet* object (the Google documentation refers to *Spreadsheet* as a “class”). It provides a method that returns an object representing the active sheet (*getActiveSheet()*) and that this returned object has a method that returns a range (*getRange()*), in this instance a single cell with the address “A2”. The returned range object method, *setValue()*, is then called with a string argument that is written to cell A1 of the active sheet. These types of chained method calls look daunting at first. The method call chain described above could be re-written as:

```
1  var greeting = 'Hello world!',
2      activeSpreadsheet =
3      SpreadsheetApp.getActiveSpreadsheet(),
4      activeSheet =
5      activeSpreadsheet.getActiveSheet(),
6      rng = activeSheet.getRange('A1'),
7      greeting = 'Hello world!';
8  rng.setValue(greeting);
```

The code above uses a number of intermediate variables and may be easier to understand initially but after experience with GAS, the chained method call will start to feel easier and more natural. The objects referenced in this example will be discussed in detail in chapters 4 and 5.

Secondly, the example code shows how easy it is to call another service from a Google Apps Script function. Here the *LanguageApp* was used to translate a simple text message into Spanish, German, and French. This ability to seamlessly access other Google services is extremely powerful.

## 2.3 Summary

This chapter has shown how to access the GAS Script Editor and execute functions from it. The examples demonstrated how GAS can display simple alerts, write messages to the *Logger*, manipulate ranges in spreadsheets and use other Google applications such as *Document* and Google services such as *LanguageApp*. These examples barely scrape the surface of what can be achieved using GAS. The next chapter uses GAS to write user-defined functions that can be called in the same manner as built-in spreadsheet functions.

# Chapter 3: User-Defined Functions

## 3.1 Introduction

User-defined functions allow spreadsheet users and developers to extend spreadsheet functionality. No spreadsheet application can cater for all requirements for all users so a mechanism is provided that allows users to write their own customised functions in the hosted language, be that VBA in Excel or GAS in Google Sheets.



### Definition

A user-defined function is one that can be called using the *equals* (=) sign in a spreadsheet cell and writes its return value or values to the spreadsheet.

User-defined functions are also known as *custom functions* and this is the term used by the Google.



### Google Custom Function Documentation

[Worth Reading!](#)<sup>7</sup>.

The source code for this chapter can be found [here on GitHub](#).<sup>8</sup> I have adopted the convention of writing user-defined functions in

---

<sup>7</sup><https://developers.google.com/apps-script/guides/sheets/functions>

<sup>8</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015/blob/master/ch03.](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch03.gs)

all upper case, other functions that are not intended to be called as user-defined functions are written in camel case. This chapter is quite long and covers a lot of material. It is also the chapter where I put most effort into explaining basic JavaScript concepts such as functions, arrays, objects and so on. Therefore, if you are unfamiliar with GAS/JavaScript, then pay close attention to the code examples and explanatory text. If my explanations are inadequate, then you can take advantage of the extensive on-line JavaScript resources available for all levels of user in addition to the many excellent JavaScript textbooks (see Appendix B for information on resources that I have found useful).

## 3.2 Built-in Versus User-Defined Functions

Modern spreadsheet applications, including Google Sheets, supply a large number of built-in functions and they owe much of their utility and widespread usage in diverse domains of finance, science, engineering and so on to these functions. There is also a high degree of standardisation between spreadsheet applications regarding function names, arguments, and usage so that they generally work uniformly in Google Sheets, Microsoft Excel, OpenOffice Calc and Gnumeric. Built-in Google Sheets functions are spread over multiple categories, see [Google Spreadsheet function list](#)<sup>9</sup>. Most of the standard spreadsheet text, date, statistical, logic and lookup functions are present and their usage is, in all cases that I have encountered, identical to equivalents in other spreadsheet applications.

Google Sheets also implements a number of novel functions. Some of these are very convenient and it is worth being familiar with them before you embark on writing your own functions so that you do not end up implementing something that is already present. One

---

<sup>9</sup><https://support.google.com/drive/bin/static.py?hl=en&topic=25273&page=table.cs>

of my favourites is the *QUERY* function and a good description of its syntax and use can be found [here](#)<sup>10</sup>. This function comes from the Google Visualization API and it uses an SQL-like syntax to extract sub-sets of values from input spreadsheet data cells. If you are already familiar with SQL, then you will feel right at home with this function. Another group of functions worth exploring are those that use **regular expressions**. The functions concerned are *REGEXMATCH*, *REGEXEXTRACT* and *REGEXREPLACE* and I have described them in a [blog entry](#)<sup>11</sup>. Regular expressions are very useful and GAS implements the full standard JavaScript regular expression specification.



## New Functionality

Google Sheets continues to add new functions

### 3.3 Why Write User-Defined Functions

The two main reasons for writing user-defined functions are for clarity and to extend functionality. User-defined functions can add to clarity by wrapping complex computations in a named and documented function. Spreadsheet power-users can often ingeniously combine the built-in functions to effectively create new ones. The disadvantage of this approach is that the resulting formulas can be very difficult to read, understand and debug. When such functionality is captured in a function, it can be given a name, documented and, tested. Secondly, user-defined functions allow developers to customise spreadsheet applications by adding functionality for their particular domain.

---

<sup>10</sup><https://anandexcels.wordpress.com/2013/11/01/query-function-in-google-sheets/>

<sup>11</sup><http://www.javascript-spreadsheet-programming.com/2013/09/regular-expressions.html>



## 3.4 What User-Defined Functions Cannot Do

An important point about user-defined functions is that they cannot be used to alter any properties, such as formats, of the spreadsheet or any of its cells. They cannot be used to send e-mails and cannot be used to insert new worksheets. Functions can be used to do these things but they cannot be called as user-defined functions. This is a common area of misunderstanding where users attempt to call functions from the spreadsheet using the *equals* operator (=). This results in an error because the function is trying to set some spreadsheet property.

User-defined functions should be designed to work just like built-in functions in that you pass in zero or more values as arguments and they return a value or values in the form of an array. Their purpose is their return values, not their side effects. Excel VBA makes a distinction between subroutines and functions. Subroutines do not return a result and cannot be called as user-defined functions. In GAS we only have functions that either return values or do not (void functions).

To illustrate how a user-defined function cannot alter spreadsheet properties, consider the following function that takes a range address argument (a string, not a *Range* object) and sets the font for the range to bold:

### Code Example 3.1

```

1  /**
2  * Simple function that cannot be called from
3  * the spreadsheet as a user-defined function
4  * because it sets a spreadsheet property.
5  *
6  * @param {String} rangeAddress
7  * @return {undefined}
8  */
9  function setRangeFontBold (rangeAddress) {
10     var sheet =
11         SpreadsheetApp.getActiveSheet();
12     sheet.getRange(rangeAddress)
13         .setFontWeight('bold');
14 }

```

The code uses objects that have not yet been discussed but the idea is simple; Take a range address as input and set the font to bold for that range. However, when the function is called from a spreadsheet, it does not work.

	A	B	C	D	E	F
1	name	department	phone			
2	Jones	Sales	1234			
3	Sanchez	Sales	6789	#ERROR!		
4	Grover	Accounting	3456			
5	Patel	Sales	7891			
6						
7						
8						
9						
10						

Figure 3-1: Error displayed when calling a function with side effects from a spreadsheet.

The error shown above refers to permissions but the problem is that the called function is attempting to modify sheet properties.

To prove that the function `setRangeFontBold()` is valid, here is a function that prompts for a range address using an *prompt* dialog. It calls the `setRangeFontBold()` function passing the given range address as an argument.



## To See The Prompt

Call the function in the Script Editor Then switch to the spreadsheet

### Code Example 3.2

```
1  /**
2  * A function that demonstrates that function
3  * "setRangeFontBold() is valid although it
4  * cannot be called as a user-defined function.
5  *
6  * @return {undefined}
7  */
8  function call_setCellFontBold () {
9      var ui = SpreadsheetApp.getUi(),
10         response = ui.prompt(
11             'Set Range Font Bold',
12             'Provide a range address',
13             ui.ButtonSet.OK_CANCEL),
14         rangeAddress = response.getResponseText();
15         setRangeFontBold(rangeAddress);
16     }
```

When this function is called, the following prompt is displayed in the spreadsheet view:

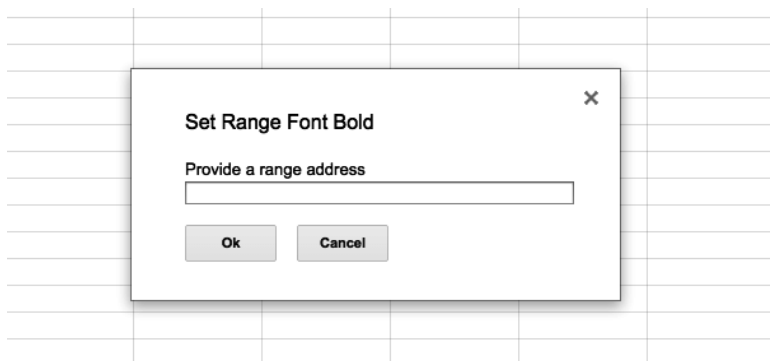


Figure 3-2: \*Prompt dialog display for a range address.

The *prompt* mechanism for requesting user input will be discussed in a later chapter. The important point here is that a function may be valid but, if it has side effects such as altering range properties, then it cannot be called as a user-defined function.



## New Sheets Feature

*Browser.inputBox* instead of *prompt()* is used in older versions and still works.

## 3.5 Introducing JavaScript Functions

User-defined functions can be called exactly like built-in ones using the *equals* (=) operator in a spreadsheet cell. They can take zero or more arguments of different types and these arguments can be either cell references or literal values, just as with built-in functions. The user-defined functions themselves are written in GAS and, as I have repeated multiple times already, GAS is JavaScript. To understand and write user-defined functions in Google Sheets requires an understanding of JavaScript functions. The better you understand JavaScript functions, the better you will be at crafting your own user-defined GAS functions for Google Sheets!

JavaScript functions are immensely powerful and flexible. They play a central role in many JavaScript idioms and patterns and mastery of them is a prerequisite to becoming an advanced JavaScript programmer. Only the basics of JavaScript functions are required for the purposes of this chapter. There are different ways of defining JavaScript functions but in this chapter most of the functions are defined as **function declarations** in form:

```
1 function functionName(comma-separated
2                       parameters) {
3   statements
4 }
```

Other types of function definition will be described as they are introduced later in the book.



## JavaScript Functions

[Read this Mozilla resource<sup>12</sup>](#)

The Mozilla resource above discusses various means of defining JavaScript functions. For user-defined functions the decision on the syntax to use to define our functions is made for us because only function declarations work in this setting, **named function expressions** and **object methods** will not work.

For the purposes of this chapter, here are the principal additional points of note regarding JavaScript functions:

- When the function is called, the arguments passed in are assigned to the parameters in the function definition.
- These arguments can then be used within the function body.

---

<sup>12</sup><https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions>

- Parameters are always passed by value in JavaScript but when reference types, such as arrays, are used, the behaviour can appear pass-by-reference.
- If the number of arguments passed in is less than the number of parameters in the function definition, the unassigned parameters are given the value *undefined*.
- Variables declared within the function using the *var* statement are local. That means that they are scoped to that function and are not visible outside the function.
- Functions in JavaScript are objects and they define a number of pre-defined properties.
- Two function properties are important for this chapter. These are the *arguments* object and the *length* property.
- The *arguments* object is an array-like list that stores all the arguments passed to the function when it is called.
- The *arguments* object is not an array but, like the *Array* type, it does have a *length* property that stores the number of arguments that were actually given when the function was called and its elements can be accessed using array-like indexing
- The *length* property of the function stores the number of arguments the function expects based on the number of parameters in the function definition.
- Since JavaScript functions can accept any number of arguments of any type regardless of the parameter list, the *arguments.length* value can be compared to the function *length* property to check if the argument count is as expected.
- Functions can have an explicit *return* statement. If no *return* statement is specified, the function will return the value *undefined*.
- User-defined functions without a *return* statement are pointless.
- A *return* statement on its own can be used to exit the function

and will return *undefined*, again, not much use in user-defined functions.

- User-defined functions should always return some value other than *undefined*.
- The returned value can be a primitive such as a string, Boolean, or a number. It can also be a reference type such as a JavaScript object or an array.

To see some of these points in action, paste the following code example 3.3 into the Script Script editor and choose and run the second function named *call\_testFunc*. The point of this example is to show how the number of function arguments and the argument data types can be determined in JavaScript.

### Code Example 3.3

```
1  /**
2   * Function to demonstrate how to check
3   * the number and types of passed arguments.
4   *
5   *
6   * @return {undefined}
7   */
8  function testFunc(arg1, arg2) {
9     var i;
10    Logger.log('Number of arguments given: ' +
11              arguments.length);
12    Logger.log('Number of arguments expected: ' +
13              testFunc.length);
14    for (i = 0; i < arguments.length; i += 1) {
15        Logger.log('The type of argument number ' +
16                  (i + 1) + ' is ' +
17                  typeof arguments[i]);
18    }
19 }
```

The function output demonstrates how JavaScript functions can check the number of arguments passed in and the type of each argument by:

1. Checking the argument count given when the function is called (*arguments.length*) against the argument count expected based on the parameter list in the function definition (the function *length* property).
2. Using the *typeof* operator to determine the type of each given argument. The *typeof* operator can also be used to identify missing argument values where the missing value will be of type *undefined*. This check can be used to assign defaults using the statement *if (typeof arg === 'undefined') { arg = default; }*.

## 3.6 User-Defined Functions Versus JavaScript Functions

When user-defined functions are called from within spreadsheet cells, their string arguments **must be provided in double quotes**. In JavaScript, either single or double quotes can be used to enclose a string literal but only double quotes are acceptable for string literals in user-defined functions. Single quote usage can lead to puzzling errors so beware!

JavaScript function names and variable names are case-sensitive, *function afunc () { ... }* and *function aFunc () { ... }* are two different functions (unlike VBA). However, when called from a spreadsheet, the function names are **case-insensitive**. The implication of this is to be very careful when naming functions to be called from a spreadsheet. Camel casing is the JavaScript standard and is used throughout here except when defining testing functions that may be given the prefix “testing”/”run” or when writing user-defined functions where the function names are all in uppercase.



## 3.7 Using JSDoc To Document Functions

JSDoc is a markup language, inspired by its Java equivalent called Javadoc, for adding comments and structured annotation to JavaScript code. A sub-set of JSDoc can be used to document user-defined functions. The JSDoc markup can then be extracted using various tools to auto-generate documentation. It is defined by a special type of multi-line commenting where the opening comment tag is defined with a special opening tag of a forward slash followed by a double asterisk:

```
1  /**
2   * Line describing the function
3   * @param {data type} parameter name
4   * @return {data type}
5   * @customfunction
6   */
```

The documentation example above contains a description line and three types of annotation tags denoted by a leading “@”. Each function parameter should be assigned an *@param* tag denoting the expected data type and the parameter name. Unsurprisingly, the return data type is given the *@return* tag. The final tag is called *@customfunction* and it is used to signify that the function is intended to be called as a user-defined function. When JSDoc is used with this tag, the user-defined functions appear in the autocomplete on entering the function name preceded by an equals sign in a spreadsheet cell.

## 3.8 Checking Input And Throwing Errors

JavaScript functions are flexible and can be called with any number of arguments of any type. Stricter languages offer some protection

against bad argument values. Most languages throw errors if the passed argument count does not match the parameter count while static languages such as Java check the given data types of function and method arguments. JavaScript takes a very relaxed approach but inappropriate argument types can cause errors or, more seriously, bugs where the return value may not be what you expect. For example:

#### Code Example 3.4

```
1 // Function that is expected
2 // to add numbers but will also
3 // "add" strings.
4 function adder(a, b) {
5     return a + b;
6 }
7 // Test "adder()" with numeric arguments
8 // and with one numeric and one string
9 // argument.
10 function run_adder() {
11     Logger.log(adder(1, 2));
12     Logger.log(adder('cat', 1));
13 }
```

The problem here is twofold. Firstly, the *adder()* function takes arguments of any type. Secondly, the *+* operator can do string concatenation as well as numeric addition. When one of the operands is a string, JavaScript *silently* coerces the other argument to type *string*. Although this example is contrived, it does illustrate a potential source of bugs. One way to guard against this is to explicitly check each argument type and then throw an error if any of the arguments is of the wrong type. The *typeof* operator can be used to do this. Here is a version of the *adder()* function that does exactly this:

#### Code Example 3.5

```
1 // Function that is expected
2 // to add numbers but will also
3 // "add" strings.
4 function adder(a, b) {
5     return a + b;
6 }
7
8 // Function that checks that
9 // both arguments are of type number.
10 // Throws an error if this is not true.
11 function adder(a, b) {
12     if (!(typeof a === 'number' &&
13         typeof b === 'number')) {
14         throw TypeError(
15             'TypeError: ' +
16             'Both arguments must be numeric!');
17     }
18     return a + b;
19 }
20
21 // Test "adder()" with numeric arguments
22 // Thrown error is caught, see logger.
23 function run_adder() {
24     Logger.log(adder(1, 2));
25     try {
26         Logger.log(adder('cat', 1));
27     } catch (error) {
28         Logger.log(error.message);
29     }
30 }
```

Now the function `adder()` throws an error if either of the arguments is non-numeric. The test function `run_adder()` uses a `try .. catch` construct to deal with the error. Detecting and dealing with incorrect arguments in functions may or may not be a priority depending

on circumstances. The *typeof* operator is adequate for primitive data types such as numbers and strings but limited for objects. For example, it reports arrays as type *object*, true but not very useful. When passing objects to functions, the *instanceof* operator is better suited for type checking.

## 3.9 Encapsulating A Complex Calculation

The **relative standard deviation**<sup>13</sup> (RSD) is frequently used in statistics to express and compare variability of data sets. It is not provided as a built-in spreadsheet function. The formula to calculate it is simply the sample standard deviation divided by the sample mean multiplied by 100. Given the following values in cells A1 to A10: 19.81 18.29 21.47 22.54 20.17 20.1 17.61 20.91 21.62 19.17 The RSD rounded to two decimal places can be calculated using this spreadsheet formula:

```
1 =ROUND(100*(STDEV(A1:A10)/AVERAGE(A1:A10)),2)
```

The functionality expressed in this spreadsheet formula can be encapsulated in a user-defined function as follows:

### Code Example 3.6

---

<sup>13</sup>[http://en.wikipedia.org/wiki/Relative\\_standard\\_deviation](http://en.wikipedia.org/wiki/Relative_standard_deviation)

```

1  /**
2   * Given the man and standard deviation
3   * return the relative standard deviation.
4   *
5   * @param {number} stdev
6   * @param {number} mean
7   * @return {number}
8   * @customfunction
9   */
10 function RSD (stdev, mean) {
11   return 100 * (stdev/mean);
12 }

```

This function can be called as follows:

```
1 =RSD(STDEV(A1:A10), AVERAGE(A1:A10))
```

	A	B	C	D	E
1	values				
2	19.81	=RSD			
3	18.29	RSD			
4	21.47	Given the man and standard deviation return the relative st...			
5	22.54				
6	20.17				

Figure 3-3: \*Calling the RSD function in Google Sheets with provided auto-complete.

The above function can be pasted into the Script Editor and called as described in the documentation. So what was gained by writing a user-defined function when a combination of spreadsheet built-in functions can be used and the user-defined version still uses the spreadsheet STDEV and AVERAGE built-ins? Firstly, the calculation now has a meaningful name. Secondly, argument checking can be added as described in the previous section. Lastly, the logic is captured in one place and is documented. These are all good reasons

to consider replacing complex spreadsheet formulas with user-defined functions. In addition, our JSDoc description now appears in the autocomplete.

## 3.10 Numeric Calculations

The following example takes a single numeric argument as degrees Centigrade and uses that argument to return the temperature in Fahrenheit. To use, paste the function into the Google Apps Script editor and then call it by name in a cell in the containing spreadsheet. For example, if cell A1 contains the value 0, then the formula `=CELSIUSTOFAHRENHEIT(A1)` in cell B1 will return the result 32 as expected.

### Code Example 3.7

```
1  /**
2   * Given a temperature in Celsius, return Fahrenheit va\
3   lue.
4   *
5   * @param {number} celsius
6   * @return {number}
7   * @customfunction
8   */
9  function CELSIUSTOFAHRENHEIT(celsius) {
10   if (typeof celsius !== 'number') {
11     throw TypeError('Celsius value must be a number');
12   }
13   return ((celsius * 9) / 5) + 32;
14 }
```

Function `CELSIUSTOFAHRENHEIT` takes a numeric value representing degrees Celsius and returns the Fahrenheit equivalent. Before performing the calculation, it first checks that the given value is

numeric, if not, it throws a type error. Here is a screenshot showing some input values and the outputs after calling this function:

	A	B	C	D
fx		=CELSIUSTOFAHRENHEIT(A9)		
1	Celsius	CELSIUSTOFAHRENHEIT		
2	0	32		
3	37	98.6		
4	100	212		
5	very hot	#ERROR!		
6		#ERROR!		
7	4/16/2015	#ERROR!		
8	very hot	#ERROR!		
9	10	50		
10				

Figure 3-4: Input and output for user-defined function *CELSIUSTOFAHRENHEIT* and .

See how the thrown error is reported as a comment in cells A5 to A8. Without the *typeof* check, the date value in cell A7 would be interpreted as a number of seconds and would return an unfeasibly large value. The outcome of evaluating the cell A6 is even more insidious because the **blank is interpreted as zero** and the function evaluates to 32. Indeed, 0 Celsius is equal to 32 Fahrenheit but this is highly misleading. Remove or comment out the *typeof* check and see the effect! The text value in cell A5 will throw an error regardless of the *typeof* check, which is to be expected. However, the default interpretation of dates as numbers and blank cells as zero is a serious trap for the unwary. The same problem occurs in Excel VBA and is more difficult to guard against. This example of a user-defined function is very simple but it highlights the need for rigorous checking of inputs.

Verifying that arguments are within a valid range may also be required to avoid invalid return values. As an example, the formula to calculate the area of a circle is simple:  $\pi * radius^2$ . The following function does this but it also checks that the given argument is both

numeric and not less than zero.

### Code Example 3.8

```
1  /**
2   * Given the radius, return the area of the circle.
3   * @param {number} radius
4   * @return {number}
5   * @customfunction
6   */
7  function AREAOFCIRCLE (radius) {
8    if (typeof radius !== 'number' || radius < 0){
9      throw Error('Radius must be a positive number');
10   }
11   return Math.PI * (radius * radius);
12 }
```

In example 3.8, the function expects a single numeric argument for the circle radius. However, since a negative radius makes no sense in this context, there is an additional check to ensure that the radius given is not negative. If either of these conditions fails, an error is thrown (`||` is the *or* operator). In this example, both checks were performed together using the *or* operator. If greater granularity is required, the checks could be performed in two separate checks that would yield two different errors. This approach helps when functions are part of larger applications.

## 3.11 Date Functions

Google Sheets provides a comprehensive set of date functions, see [here](#)<sup>14</sup> for the full list. However, if the provided functions do not cover your requirements, you can write your own in GAS and

---

<sup>14</sup><https://support.google.com/docs/table/25273?hl=en&rd=2>



call them from your spreadsheet as user-defined functions. Some cautionary words: Dates and times can be quite complicated in any application or programming language where you have to allow for time zones, daylight saving time changes and leap years so rigorous testing of inputs and outputs is highly advisable.

The examples given here also cover some important basic JavaScript concepts such as date manipulation, objects and methods and JavaScript arrays.

The first example (3.9) takes a date as its sole argument and returns the day name (in English) for that date.

### Code Example 3.9

```
1  /**
2   * Given a date, return the name of the
3   * day for that date.
4   *
5   * @param {Date} date
6   * @return {String}
7   * @customfunction
8   */
9  function DAYNAME(date) {
10     var dayNumberNameMap = {
11         0: 'Sunday',
12         1: 'Monday',
13         2: 'Tuesday',
14         3: 'Wednesday',
15         4: 'Thursday',
16         5: 'Friday',
17         6: 'Saturday'},
18         dayName,
19         dayNumber;
20     if(! date.getDay ) {
21         throw TypeError(
```

```
22     'TypeError: Argument is not of type "Date"!');
23   }
24   dayNumber = date.getDay();
25   dayName = dayNumberNameMap[dayNumber];
26   return dayName;
27 }
```

The most interesting points in the code example 3.9 are the use of the object named *dayNumberNameMap*, how the argument type is checked and how the *Date* object method *getDay()* is used. The variable *dayNumberNameMap* is a JavaScript object. Since objects are cornerstones of JavaScript programming, here is a short detour that describes them. At their simplest, JavaScript objects provide a mapping of keys to values. The values are called “properties” and they can be other objects, scalar values, arrays or functions. When object properties are functions, they are known as **methods**. JavaScript objects are analogous to hashes or dictionaries in other languages. The very popular data interchange format called JavaScript Object Notation (JSON) is based on them. Excel VBA offers a *Dictionary Object* from the [Microsoft Scripting Library](#)<sup>15</sup> that provides similar functionality but is much less flexible. The object in the example above maps index numbers to the names of the days of the week. Returning to the code example above, the *date* argument is expected to be of type *Date* and the mechanism used to test for it, that is the line *if(! date.getDay ) {*, merits some explanation. If the argument is a *Date* object, then it will implement a *getDate()* method. The test here is to see if the property exists, notice how the *getDate()* method is not called because there are no parentheses. If the argument does not have a *getDate* property, then an error is raised. This type of checking for methods on objects is common in client-side (browser) JavaScript where implementations from different vendors may or may not provide certain methods. If the argument passes the type check, then we are satisfied that it is a

---

<sup>15</sup><https://support.microsoft.com/en-us/kb/187234/>

*Date* object so we can call the *getDay()* method on it. The *getDay()* method returns the week day as an integer starting with 0 (zero) for Sunday. Having the day number, we can use it as key for our object *dayNumberNameMap* to return the day name. To use the function *DAYNAME* as a user-defined function, copy the code into the Script Editor and call it from the spreadsheet like this where the return value of the built-in *TODAY()* function is given as an argument:

```
1 =DAYNAME(TODAY())
```

All the user-defined functions given so far return a scalar value but they can also return arrays where each array element is entered into its own cell. An example will help to illustrate this. In this example, we write a function that returns all dates that fall on a given day between two dates.

### Code Example 3.10

```
1  /**
2  * Add a given number of days to the given date
3  * and return the new date.
4  *
5  * @param {Date} date
6  * @param {number} days
7  * @return {Date}
8  */
9  function addDays(date, days) {
10     // Taken from Stackoverflow:
11     // questions/563406/add-days-to-datetime
12     var result = new Date(date);
13     result.setDate(result.getDate() + days);
14     return result;
15 }
16
17 /**
```

```
18 * Given a start date, an end date and a day name,
19 * return an array of all dates that fall (inclusive)
20 * between those two dates for the given day name.
21 *
22 * @param {Date} startDate
23 * @param {Date} endDate
24 * @param {String} dayName
25 * @return {Date[]}
26 * @customfunction
27 */
28 function DATESOFDAY(startDate, endDate, dayName) {
29     var dayNameDates = [],
30         testDate = startDate,
31         testDayName;
32     while(testDate <= endDate) {
33         testDayName = DAYNAME(testDate);
34         if(testDayName.toLowerCase() ===
35            dayName.toLowerCase()) {
36             dayNameDates.push(testDate);
37         }
38         testDate = addDays(testDate, 1);
39     }
40     return dayNameDates;
41 }
```

The code in example 3.10 defines a user-defined function called *DATESOFDAY()* that calls two other GAS functions, *addDays()* and *DAYNAME()* to perform some of the computation. Function *DAYNAME()* was described in code example 3.9 above will be needed when you paste this code into the Script Editor. The function *addDays()* was copied from this [location on Stackoverflow](http://stackoverflow.com/questions/563406/add-days-to-datetime).<sup>16</sup> I stated earlier that date manipulation can be complex and the discussion thread on this Stackoverflow entry certainly proves that.

---

<sup>16</sup><http://stackoverflow.com/questions/563406/add-days-to-datetime>

Before discussing the actual code in example 3.10 in detail, first note how a user-defined function can call other GAS functions to perform some of its tasks as can be seen in this example where two other functions are called and their return values are used by *DATESOFDAY()*. By off-loading much of its work, *DATESOFDAY()* remains small on manageable.



## Make Functions Small

De-composing a programming task into multiple small functions makes each function easier to test, de-bug and re-use

*DATESOFDAY()* takes two dates and a day name as argument. I have not added any checking of types or numbers of argument, these could be easily added using the techniques discussed earlier. The function declares two local variables, *dayNameDates* and *testDate*. It initialises *testDate* to *startDate* and then enters the *while* loop. Each round of the *while* loop gets the name of the day for the *testDate* using the *DAYNAME* function described in code example 3.9. The returned day name is then compared to the given argument day name (*dayName*) using a **case-insensitive** comparison by calling the string method *toLowerCase()* on both string operands. If they are equal, the date is added to the array using the *Array push()* method. This technique of building arrays in loops will be seen throughout the book. JavaScript arrays are extremely flexible and implement a range of very useful methods. Many of these methods will be encountered later in this book. After testing the day of the date, the test date is incremented by one day using the *addDays()* function. The condition *testDate <= endDate* is tested in the next round of the *while* loop and the loop finishes when the condition evaluates to *false* and the *dayNameDates* array is returned.

When the function *DATESOFDAY()* is called as a user-defined

function, it writes the entire array to a single column, see the screenshot below:

fx   =DATESOFDAY(A2,A3,A4)			
	A	B	C
1	<b>InputParameters</b>	<b>ArrayOutput</b>	
2	4/17/2015	4/21/2015	
3	6/1/2015	4/28/2015	
4	Tuesday	5/5/2015	
5		5/12/2015	
6		5/19/2015	
7		5/26/2015	
8			

Figure 3-5: Calling user-defined function that returns an array of values.

## 3.12 Text Functions

Google Spreadsheets text manipulation functions can also be complemented and enhanced by writing user-defined functions. JavaScript, like most programming languages, refers to the text type as “string”. Strings in JavaScript are primitive types but they can be wrapped to behave like objects. The wrapping of a primitive so that it becomes an object is known as **auto-boxing**. Auto-boxing can also be applied to *Boolean* and *Number* primitive types and though I have never seen any use for it, with *Boolean*. The *Number* type has some useful methods like *toFixed()*. Since auto-boxing is transparent to the user/programmer, all that really matters is that strings can be treated as though they have methods that can be used to extract information from them and to generate new strings.

To see all the String properties supported by GAS/JavaScript, paste the following code into the script editor and execute it. An alphabetical list of string properties is sent to the logger.

### Code Example 3.11

```
1  /**
2   * Print all string properties to the
3   * Script Editor Logger
4   * @return {undefined}
5   */
6  function printStringMethods() {
7      var strMethods =
8          Object.getOwnPropertyNames(
9              String.prototype);
10     Logger.log('String has ' +
11               strMethods.length +
12               ' properties.');
```

```
13     Logger.log(strMethods.sort().join('\n'));
14 }
```

The code above uses JavaScript’s very powerful introspection capabilities to extract an array of all the property names defined for object *String* on what is known as its **prototype**. The prototype is a property that refers to an object where the *String* methods are defined in the prototype. Prototypes are quite an advanced topic but they are very important and provide the basis for inheritance in JavaScript. All JavaScript objects (that includes arrays and functions) have a prototype property. The logger output shows that there are *38 String* properties. Most are methods but the important *length* property, for example, is not. Code example 3.11 also uses two important *Array* methods: *sort()* and *join()*. The *sort()* method does an alphabetical sort in this example and *join()* then makes a string of the array using the given argument (new line ‘\n’ in this example) to separate the concatenated array elements. Notice how the method calls are “chained” one after the other. This is a common approach in JavaScript. As an exercise, you could try writing an equivalent function to print out all the array properties.

Since the *String* object has so many methods, we can use these methods in user-defined functions. Here is a function that returns

the argument string in reverse character order.

### Code Example 3.12

```
1  /**
2   * Given a string, return a new string
3   * with the characters in reverse order.
4   *
5   * @param {String} str
6   * @return {String}
7   * @customfunction
8   */
9  function REVERSESTRING(str) {
10   var strReversed = '',
11     lastCharIndex = str.length - 1,
12     i;
13   for (i = lastCharIndex; i >= 0; i -= 1) {
14     strReversed += str[i];
15   }
16   return strReversed;
17 }
```

Reversing strings might not strike you as the most useful functionality unless of course you work with DNA or are interested in palindromes. To see a palindrome example, call the function with the word *detartrated*, yes, that word really exists. The example function uses a JavaScript *for* loop to extract the string characters in reverse order and appends them to a new string that is built within the loop. The newly built string is returned upon completion of the loop. If you declare the variable *strReversed* but do not assign it, something unexpected happens. As the variable is unassigned, it is *undefined*. If you concatenate a string to *undefined* you get another of those troublesome implicit JavaScript conversions that I warned about earlier where *undefined* becomes the string “undefined! Beware!



Spreadsheets generally offer a large set of built-in text functions and Google Sheets augments these with regular expression functions. I have covered regular expressions in two blog entries (see part 1 [here](#)<sup>17</sup> and part 2 [here](#)<sup>18</sup>.) I will include a chapter on regular expression later in this book and will give some regular expression-based user-defined function examples there.

## 3.13 Using JavaScript Built-In Object Methods

JavaScript provides a small number of built-in objects that provide additional functionality that can be used in user-defined functions. For example, the JavaScript *Math* object is available in GAS and provides some useful methods. Simulation of a die throw can be easily implemented using its methods:

### Code Example 3.13

```
1  /**
2   * Simulate a throw of a die
3   * by returning a number between
4   * and 6.
5   *
6   * @return {number}
7   * @customfunction
8   */
9  function THROWDIE() {
10   return 1 + Math.floor(Math.random() * 6);
11 }
```

---

<sup>17</sup><http://www.javascript-spreadsheet-programming.com/2013/09/regular-expressions.html>

<sup>18</sup><http://www.javascript-spreadsheet-programming.com/2014/09/regular-expressions-part-2.html>

Try out the function by calling `=THROWDIE()` from any spreadsheet cell.

This function uses two methods defined in the JavaScript built-in *Math* object to simulate dice throwing. The *Math round()* method could also be used here but, as is [very well described here](#)<sup>19</sup>, it leads to biased results. This becomes apparent when the function is called hundreds of times. The important point to note is the need for testing and reading of documentation to avoid inadvertent errors and biases.

## 3.14 Using A Function Callback

The object nature of JavaScript functions allows them to be both passed as arguments to other functions and to be returned by other functions. The following example provides an example of the power and flexibility of JavaScript. It also shows how easy it is to use a range of cells as an argument. The function concatenates an input range of cells using a delimiter argument. There is an in-built Google Spreadsheet function called JOIN that performs this task but the user-defined version below has one additional feature: It provides an option to enclose each individual element in single quotes. Here is the code:

### Code Example 3.14

---

<sup>19</sup><http://www.the-art-of-web.com/javascript/random/#.UPU8tKHC-nY>

```

1  /** Concatenate cell values from
2  * an input range.
3  * Single quotes around concatenated
4  * elements are optional.
5  *
6  * @param {String[]} inputFromRng
7  * @param {String} concatStr
8  * @param {Boolean} addSingleQuotes
9  * @return {String}
10 * @customfunction
11 */
12 function CONCATRANGE(inputFromRng, concatStr,
13                      addSingleQuotes) {
14   var cellValues;
15   if (addSingleQuotes) {
16     cellValues =
17       inputFromRng.map(
18         function (element) {
19           return "'" + element + "'";
20         });
21     return cellValues.join(concatStr);
22   }
23   return inputFromRng.join(concatStr);
24 }

```

An example of a call to this function in the spreadsheet is `=CONCATRANGE(A1:A5, ",", true)`. This returns output like `'A';B';C';D';E'`. There are two interesting aspects to this code:

1. The range input is converted to a JavaScript array. In order to work correctly, all the input must be from the same column. If the input range is two-dimensional or from multiple cells in the same row, it will generate a JavaScript array-of-arrays. That point is ignored here, but try giving it a two-dimensional range input or something like `A1:C2` and observe the output!

2. The more interesting point is the use of the *Array map()* method. This method iterates the array and applies its function argument to each element of the array and returns a new array. The function used here is an **anonymous function**. It takes one argument, the array element, and returns it enclosed in single quotes. The anonymous function operates as a **callback**. More examples of callbacks will be given in later chapters. This example just hints at the power of this approach.

A version of this function was written in Excel VBA as a quick way of generating input for the *IN* clause in SQL. Lists of string values (VARCHARs in database terminology) would be provided in spreadsheets so an Excel user-defined function was written to allow the lists to be concatenated in comma-separated strings with the constituent values enclosed in single quotes. There are better ways of running such queries that will be discussed in the JDBC chapter but this is an approach that can be useful.

It is worth noting that built-in Google Spreadsheets can be combined to get the same output:

```
1 =CONCATENATE("'", JOIN("'",',', A1:A5), "'")
```

However, no introduction to JavaScript functions would be complete without at least a brief mention of anonymous functions and callbacks.

## 3.15 Extracting Useful Information About The Spreadsheet

All the user-defined functions discussed up to this point were written in host-independent JavaScript. They should all work in modern browsers or in Node.js scripts.



## Running Examples In Node.js

Replace `Logger.log()` with `*console.log()`

The restrictions on the actions of user-defined functions were also discussed. Despite these restrictions, useful information about a spreadsheet can be returned to spreadsheet cells by means of user-defined functions. The examples given below are all functions that take no arguments and return some useful information about the active spreadsheet. It is worth noting how they all use *get* methods, any attempt to call *set* methods would not work.

### Code Example 3.15

```
1  /**
2   * Return the ID of the active
3   * spreadsheet.
4   *
5   * @return {String}
6   * @customfunction
7   */
8  function GETSPREADSHEETID() {
9      return SpreadsheetApp
10         .getActiveSpreadsheet().getId();
11  }
12
13  /**
14   * Return the URL of the active
15   * spreadsheet.
16   *
17   * @return {String}
18   * @customfunction
19   */
20  function GETSPREADSHEETURL() {
21      return SpreadsheetApp
```

```
22     .getActiveSpreadsheet().getUrl();
23 }
24
25 /**
26  * Return the owner of the active
27  * spreadsheet.
28  *
29  * @return {String}
30  * @customfunction
31  */
32 function GETSPREADSHEETOWNER() {
33     return SpreadsheetApp
34         .getActiveSpreadsheet().getOwner();
35 }
36
37 /**
38  *Return the viewers of the active
39  * spreadsheet.
40  *
41  * @return {String}
42  * @customfunction
43  */
44
45 function GETSPREADSHEETVIEWERS() {
46     var ss =
47         SpreadsheetApp.getActiveSpreadsheet();
48     return ss.getViewers().join(', ');
49 }
50
51 /**
52  * Return the locale of the active
53  * spreadsheet.
54  *
55  * @return {String}
56  * @customfunction
```

```
57  */
58  function GETSPREADSHEETLOCALE() {
59    var ss = SpreadsheetApp.getActiveSpreadsheet();
60    return ss.getSpreadsheetLocale();
61  }
```

These examples use GAS host objects and methods. The objects in the code examples are discussed extensively in the following two chapters, so read ahead if you wish to understand them right now. The purpose here is to just show how user-defined functions can be utilised to extract information about a spreadsheet using host object methods.

All the functions given above can extract the required information and return it in a single statement. The functions *GETSPREADSHEETVIEWERS()* and *GETSPREADSHEETLOCALE* use a variable to reference the active spreadsheet and the value returned using a method call to the variable. The reason for doing this here was simply to make the code more readable by avoiding a wrapped line in the book text.

To see these functions in action, just paste the code above into any Google Spreadsheet Script Editor and then invoke each of them from the spreadsheet using the usual syntax, example: *=GETSPREADSHEETID()*. All of the examples except *GETSPREADSHEETVIEWERS()* use a *Spreadsheet* method that returns a primitive JavaScript value, namely a string. The *getUsers()* method however returns an array and the function uses the *Array* type *join()* method to convert the array to a string. The *join()* call could be omitted and the function would still work. However, each user would be written to a separate cell. To experiment, remove the *join()* and then call the function.

## 3.16 Using Google Services

Some Google Services can be used in user-defined functions. The following example is taken from the Google Language service.

User-defined functions can be used as a sort of dictionary to translate words and phrases. The following function can all be called from the spreadsheet to perform translations from English to French.

### Code Example 3.16

```
1  /**
2   * Return French version
3   * of English input.
4   *
5   * @param {String} input
6   * @return {String}
7   */
8  function ENGLISHTOFRENCH(input) {
9    return LanguageApp
10     .translate(input, 'en', 'fr');
11 }
```

Example invocation:

```
1 =ENGLISHTOFRENCH("Try closing the file!")
```

The output: “Essayez de fermer le fichier!”

If you wish to translate into, for example, Spanish or German, replace “fr” with “es” and “de”, respectively. These functions appear to work well for single words and straight-forward, non-idiomatic, phrases. However, only a fluent speaker could comment on the translation quality. A good check is to re-translate back into the



original and see if it makes sense. I have tried some idiomatic input such as “JavaScript rocks!” and “JavaScript sucks!” - translated into standard English as “JavaScript is really great” and “JavaScript is very bad!” - and the translations, unsurprisingly, missed the point.

## 3.18 Summary

- Google Spreadsheets provides a large number of built-in functions.
- User-defined functions can be written to complement these functions and to provide new functionality.
- Before writing a user-defined function, ensure that Google Spreadsheets does not already provide an equivalent.
- User-defined functions cannot be used to set cell properties or display dialogs.
- User-defined functions are written in the Google Apps Scripting version of JavaScript.
- JavaScript functions are very powerful and very flexible.
- User-defined functions should be rigorously tested.
- There should be code to check and verify the number and type of arguments that are passed when the user-defined function is called. The JavaScript *typeof* and *instanceof* operators can be used for these purposes.
- A variety of invalid data should be passed as arguments to ensure that the user-defined function performs as expected by throwing an error.
- Use the *throw* statement with an error object (example *TypeError*) with a message for the error. Calling code can then use *try catch* statements to deal with the error.
- Ensure that the user-defined function deals appropriately with blank cells.
- Ensure that user-defined functions expecting numeric arguments do not misinterpret date input.

- Watch out for implicit JavaScript or spreadsheet type conversions.
- When using JavaScript built-in functions or methods provided by the *Math* library for example, read the documentation and test function output against expected results to avoid errors and biases.
- The object nature of JavaScript functions allows them to be used as callbacks.
- Useful information about a Google Spreadsheet document can be retrieved with user-defined functions that call methods of the *SpreadsheetApp* class.
- User-defined functions can be used to query Google services such as *LanguageApp* in order to return useful information to a spreadsheet.

# Chapter 4: Spreadsheets and Sheets

Google Spreadsheet programming revolves around manipulation of the *SpreadsheetApp*, *Spreadsheet*, *Sheet* and *Range* objects. *SpreadsheetApp*, *Spreadsheet*, and *Sheet* are the subject of the current chapter while ranges are discussed in the following one. The material in this and the following chapter is absolutely central to all subsequent content so there are lots of code examples with extensive descriptions and notes to emphasise their importance

All code for this chapter is available [here on GitHub](#)<sup>20</sup>.

## 4.1 A Note On Nomenclature

Since JavaScript is class-free, talking of classes may cause confusion. Google refers to *SpreadsheetApp*, *Spreadsheet*, etc. as classes. These are Google implementation details that are not of concern here. For *SpreadsheetApp*, no instance objects are made so it and other top-level App objects will be referred to as classes but *Spreadsheet* and all other classes within the *SpreadsheetApp* hierarchy will be referred to as objects because all the code examples given here operate on instances of these classes.

The term **properties** is used to refer to **all** object attributes so it encompasses methods and scalar values such as the *Array length* property.

---

<sup>20</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015/blob/master/ch04.gs](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch04.gs)

## 4.2 Native And Host Objects

In order to develop GAS applications it is necessary to know both JavaScript and the Google Apps Application Programming Interfaces (APIs). Learning these APIs means becoming familiar with the Google Apps object methods.

The GAS programming environment is composed of three types of objects:

1. Objects such as *Spreadsheet* provided by the hosting environment.
2. Built-in JavaScript objects such as *Array*.
3. JavaScript objects created by the programmer.

Google documentation with example code for some objects and methods is [available here](#)<sup>21</sup>

## 4.3 A Note On Method Overloading In Google Apps Script

Java and other programming languages support method **overloading**. Overloading allows different methods in the same class to have the same name provided that their parameters differ in either type or number; That is the method **signature** differs JavaScript does not support method overloading. If the same name is used more than once for a function in the same script file, the earlier versions are simply over-written. However, the *SpreadsheetApp* and other classes defined for Google Spreadsheets implement method overloading. How this is implemented need not be of concern here but it is important to know which version of an overloaded method is called and what type, if any, it returns. For example the *Spreadsheet*

---

<sup>21</sup>[https://developers.google.com/apps-script/service\\_spreadsheet](https://developers.google.com/apps-script/service_spreadsheet)

object has a method called *insertSheet()* that has eight versions each of which is distinguished by its parameters.

```

1 function methodOverloadExample () {
2   var newSheet = SpreadsheetApp.getActiveSpreadsheet().insertSheet
3 }

```

- insertSheet() : Sheet
- insertSheet(Object options) : Sheet
- insertSheet(String sheetName) : Sheet
- insertSheet(String sheetName, Object options) : Sheet
- insertSheet(String sheetName, int sheetIndex) : Sheet
- insertSheet(String sheetName, int sheetIndex, Object options) : Sheet
- insertSheet(int sheetIndex) : Sheet
- insertSheet(int sheetIndex, Object options) : Sheet

Figure 4-1: Script Editor showing overloading of the *insertSheet()* method of the Spreadsheet object.

## 4.4 The Concept of Active Objects

A user can have multiple spreadsheets open at the same time. Each open spreadsheet can have one or more sheets. A range of variable size can be selected within a sheet. There is always one, and only one, **active object** for spreadsheets, sheets, range and cell. The selected spreadsheet is the active spreadsheet. It contains an active sheet. There is a single active range within each spreadsheet and within this active range there is one active cell. The active object can be set and retrieved via *set* and *get* methods in the various Google Sheet objects. These active objects are often the default objects in the GAS examples given here.

## 4.5 Object Hierarchies

The Google Apps objects are arranged in a hierarchical structure where the methods of one object can return other objects that, in turn, have their own methods that may also return objects. This hierarchy can be nested quite deeply where objects are nested within objects. Those familiar with the DOM in web programming or those who have used Excel VBA will be familiar with this

concept. However, for those coming to it for the first time, it can appear complex and somewhat intimidating. The following function demonstrates this nesting of objects.

#### Code Example 4.1

```
1 // Function to demonstrate the Spreadsheet
2 // object hierarchy.
3 // All the variables are gathered in a
4 // JavaScript array.
5 // At each iteration of the for loop the
6 // "toString()" method
7 // is called for each variable and its
8 // output is printed to the log.
9 function showGoogleSpreadsheetHierarchy() {
10  var ss = SpreadsheetApp.getActiveSpreadsheet(),
11      sh = ss.getActiveSheet(),
12      rng = ss.getRange('A1:C10'),
13      innerRng = rng.getCell(3, 3),
14      innerRngAddress = innerRng.getA1Notation(),
15      column = innerRngAddress.slice(0,1),
16      googleObjects = [ss, sh, rng, innerRng,
17                      innerRngAddress, column],
18      i;
19  for (i = 0; i < googleObjects.length; i += 1) {
20    Logger.log(googleObjects[i].toString());
21  }
22 }
```

Paste this code into the editor, execute *showGoogleSpreadsheetHierarchy*, and then see the output that is written to the log (menu action: *View->Logs...* in the script editor). The output from code is as follows:

```
1 Spreadsheet
2 Sheet
3 Range
4 Range
5 C3
6 C
```

Here is a step-by-step explanation of the code example 4.1 and its output:

- The *SpreadsheetApp* is the starting point. Its method *getActiveSpreadsheet()* returns a *Spreadsheet* object.
- The *Spreadsheet* *getActiveSheet()* method returns a *Sheet* object.
- The *Sheet* method *getRange()* returns a *Range* object for the given address. In this example the range object corresponding to the address A1:C10 was returned.
- This *Range* object is itself composed of ranges and these “inner” ranges can be retrieved. The *Range* *getCell()* method takes a row and column index as arguments and returns the corresponding single cell as a *Range* object.
- The *Range* object has a method called *getA1Notation()* that returns the range address as a JavaScript string.
- As discussed in chapter 3, JavaScript strings are primitive types but they are endowed with a wrapping mechanism that operates in the background to represent them as objects of type *String*. The *String* method *slice(0,1)* extracts the first character of the cell address, i.e. its column.
- The *toString()* method conveniently returns the object type for all Google Apps objects. When called on a string, it simply returns the string value.

All of the above can be summarised as:

**SpreadsheetApp** → **Spreadsheet** → **Sheet** → **Range** → **Range**  
 → **String** → **String**

Where the → symbol represents a method call to the object on the left that returns the object on the right.

Method calls can be chained to produce long and complex statements. It may be hard to believe it but the following code does actually work.

### Code Example 4.2

```

1 // Print the column letter of the third row and
2 //   third column of the range "A1:C10"
3 //   of the active sheet in the active
4 //   spreadsheet.
5 // This is for demonstration purposes only!
6 function getColumnLetter () {
7   Logger.log(
8     SpreadsheetApp.getActiveSpreadsheet()
9       .getActiveSheet().getRange('A1:C10')
10      .getCell(3, 3).getA1Notation()
11      .slice(0,1));
12 }

```

When pasted into the script editor and executed, it prints just the letter C to the log. GAS code should only ever be written like this for demonstration purposes. Carefully read the comments that precede the code and then try reading the code right-to-left and then see if you understand it!

## 4.6 SpreadsheetApp

All Google applications have a top-level class. For spreadsheets, there is the *SpreadsheetApp* class while documents and e-mail have



the *DocumentApp* and *MailApp* classes, respectively. *SpreadsheetApp* methods are used to create *Spreadsheet* objects and open spreadsheets. Objects of this class are not created, rather its methods are always called using the syntax *SpreadsheetApp.some\_method*.

## 4.7 The *Spreadsheet* Object

Every Google Sheets GAS application requires a *Spreadsheet* object. Typically, an application begins with the creation of a *Spreadsheet* object to represent an actual Google spreadsheet document. *Sheet* and *Range* objects can then be created and manipulated using its methods.

The Script Editor is a very valuable tool for learning GAS. As Figure 4-2 shows, the Script Editor can display a full property list for an object when the period (dot) is typed after the object name:



```

1 function testSpreadsheetApp () {
2   var ss = SpreadsheetApp.
3 }
4

```

The dropdown menu displays the following methods for the *SpreadsheetApp* class:

- create(String name) : Spreadsheet
- create(String name, int rows, int columns) : Spreadsheet
- flush() : void
- getActive() : Spreadsheet
- getActiveRange() : Range
- getActiveSheet() : Sheet
- getActiveSpreadsheet() : Spreadsheet
- open(File file) : Spreadsheet
- openById(String id) : Spreadsheet

Figure 4-2: The Google Apps Script Editor displaying a list of available properties for the *SpreadsheetApp* class.

### 4.7.1 Writing *Spreadsheet* Object Properties To A Google Spreadsheet

The following short piece of code illustrates some key features of Google Spreadsheet programming. It demonstrates how to extract

a complete list of properties from a Google Apps object and then writes the list to a Google Spreadsheet.

### Code Example 4.3

```
1  // Extract an array of all the property names
2  // defined for Spreadsheet and write them to
3  // column A of the active sheet in the active
4  // spreadsheet.
5  function testSpreadsheet () {
6      var ss =
7          SpreadsheetApp.getActiveSpreadsheet(),
8          sh = ss.getActiveSheet(),
9          i,
10         spreadsheetProperties = [],
11         outputRngStart = sh.getRange('A2');
12     sh.getRange('A1')
13         .setValue('spreadsheet_properties');
14     sh.getRange('A1')
15         .setFontWeight('bold');
16     spreadsheetProperties =
17         Object.keys(ss).sort();
18     for (i = 0;
19         i < spreadsheetProperties.length;
20         i += 1) {
21         outputRngStart.offset(i, 0)
22             .setValue(spreadsheetProperties[i]);
23     }
24 }
```

1	spreadsheet_properties
2	addCollaborator
3	addCollaborators
4	addEditor
5	addEditors
6	addMenu
7	addViewer
8	addViewers
9	appendRow
0	ask
1	confirm
2	copy
3	deleteActiveSheet
4	deleteColumn

Figure 4-3: Sample of output from function *testSpreadsheet()* showing some of the *Spreadsheet* object properties.

When the function above is executed, it writes all the properties of the *Spreadsheet* object (107 at time of writing) to column A of the active sheet in the active spreadsheet as shown in figure 4-3 above.

Despite the comparative brevity of code example 4.3, there is quite a lot to discuss and understand. Here are the main points:

- Create a *Spreadsheet* instance for the active spreadsheet using the method *getActiveSheet()* of the *SpreadsheetApp* class and assign it to the variable *ss*.
- Get an instance of the active sheet using the aptly named *getActiveSheet()* method of the *Spreadsheet* object and assign it to the variable *sh*.
- Declare an uninitialised loop variable, *i*.
- Create an empty array literal, *spreadsheetProperties*, to store the *Spreadsheet* object properties.
- Create a range variable called *outputRngStart* that represents cell A2 of the *Sheet* object referenced by the variable *sh* by using the *getRange()* method of the *Sheet* object.
- Set the column header in cell A1 by getting a *Range* object that represents this cell and set its value (*setValue()* method)

and its text format (*setFontWeight('bold')*). Note, no variable is used here to reference this cell.

- The *Object.keys()* method is called passing it the *Spreadsheet* instance as an argument and it returns an array of all the properties of this argument.
- The *Array.sort()* method is then called to do its default alphabetical sorting.
- A JavaScript *for* loop is used to iterate the array of *Spreadsheet* properties.
- In each iteration of the loop, the value passed to the row argument of the range *offset()* method is the newly incremented *i* integer value.
- The *setValue()* call on the *Range* object returned by the method *offset()* writes its argument, the property name, to the cell.

The *Object.keys()* method used in the example code is a JavaScript introspection method. It is well described on the [Mozilla Developer Network website](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/keys).<sup>22</sup>

The preceding code discussion may appear overly verbose but it is very important to fully understand the code presented here in order to understand later chapters. It is also a good illustration of how programming in GAS is an interplay between JavaScript and Google App objects. It is worth spending some time studying and modifying this code example to get a good feel for how it works. For example, see the effect of changing the *Range.offset()* method's second argument to *i* in place of the constant 0 that was used.

---

<sup>22</sup>[https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Object/keys](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/keys)

## 4.8 The *Sheet* Object

A *Spreadsheet* object contains one or more *Sheet* objects. *Sheet* objects can be retrieved from *Spreadsheet* objects using a number of methods. The `getActiveSheet()` is one such method and it was used in the code example 4.3 above. All the properties of *Sheet* objects could be similarly extracted and printed as they were above for the *Spreadsheet* object. For brevity, here is a function that prints all *Sheet* properties to the log:

### Code Example 4.4

```
1 // Extract, an array of properties from a
2 // Sheet object.
3 // Sort the array alphabetically using the
4 // Array sort() method.
5 // Use the Array join() method to a create
6 // a string of all the Sheet properties
7 // separated by a new line.
8 function printSheetProperties () {
9   var ss =
10     SpreadsheetApp.getActiveSpreadsheet(),
11     sh = ssgetActiveSheet();
12   Logger.log(Object.keys(sh)
13     .sort().join('\n'));
14 }
```

There are currently **79 sheet properties** (up from 72 in the earlier version of this book) and these are all methods. It is worth taking time to review this list and the [Google documentation](https://developers.google.com/apps-script/class_sheet).<sup>23</sup>

---

<sup>23</sup>[https://developers.google.com/apps-script/class\\_sheet](https://developers.google.com/apps-script/class_sheet)

## 4.9 Practical Examples Using *Spreadsheet* And *Sheet* Objects

Practice with GAS is the best way to learn it. The following code examples all do something practical and tangible. They aim to provide a good starting point for Google Spreadsheet application development. For each example, paste the given code into the script editor and execute the indicated function (usually with the “test\_” prefix). Observe how the calling function deals with any errors thrown by the called function by using JavaScript’s structured exception handling. The error handling may seem excessive for such short examples but it becomes essential when writing larger applications.

### 4.9.1 Print A List Of Sheet Names In A Spreadsheet

#### Code Example 4.5

```
1  // Call function listSheets() passing it the
2  // Spreadsheet object for the active
3  // spreadsheet.
4  // The try - catch construct handles the
5  // error thrown by listSheets() if the given
6  // argument is absent or something
7  // other than a Spreadsheet object.
8  function test_listSheets () {
9    var ss =
10     SpreadsheetApp.getActiveSpreadsheet();
11     try {
12       listSheets(ss);
13     } catch (error) {
14       Logger.log(error.message);
15     }
```

```
16 }
17 // Given a Spreadsheet object,
18 // print the names of its sheets
19 // to the logger.
20 // Throw an error if the argument
21 // is missing or if it is not
22 // of type Spreadsheet.
23 function listSheets (spreadsheet) {
24     var sheets,
25         i;
26     if (spreadsheet.toString()
27         !== 'Spreadsheet') {
28         throw TypeError(
29             'Function "listSheets" expects ' +
30             'argument of type "Spreadsheet"');
31     }
32     sheets = spreadsheet.getSheets();
33     for (i = 0;
34         i < sheets.length; i += 1) {
35         Logger.log(sheets[i].getName());
36     }
37 }
```

Execution of the `test_listSheets()` function prints the names of all sheets in the active spreadsheet to the log. The `listSheets()` function will only accept a `Spreadsheet` object and throws an error back to the caller if the argument is missing or of the wrong type. The `Spreadsheet` object has a method called `getSheets()` that returns a JavaScript array of `Sheet` objects. The `Sheet` object `getName()` method is then called and it returns the name as a string.



## How to determine the type of a Google Apps object

Call the object `'toString()'` method to return the "class" of the object.





```
23 }
24 // Given a Spreadsheet object and a sheet name,
25 // return "true" if the given sheet name exists
26 // in the given Spreadsheet,
27 // else return "false".
28 function sheetExists(spreadsheet, sheetName) {
29     if (spreadsheet.getSheetByName(sheetName)) {
30         return true;
31     } else {
32         return false;
33     }
34 }
```

The test function uses an array of strings and tests each one to determine if it is the name of a sheet in the active spreadsheet. The call to the *Spreadsheet* method *getSheetByName()* is central to the *sheetExists()* function. If the sheet name exists, the method returns a *Sheet* object, otherwise it returns the special JavaScript object *null*. In a Boolean statement *null* evaluates to *false* while a *Sheet* object is evaluated as true. However, the method *getSheetByName()* is not case-sensitive. Therefore, this method treats ‘sheet1’ and ‘Sheet1’ as the same.

### 4.9.3 Copy A Sheet From One Spreadsheet To Another

When working with multiple spreadsheets in the same application, you may need to copy one sheet from one spreadsheet to another. Code example 4.7 shows how to do this.

#### Code Example 4.7

```
1 // Copy the first sheet of the active
2 // spreadsheet to a newly created
3 // spreadsheet.
4 function copySheetToSpreadsheet () {
5     var ssSource =
6         SpreadsheetApp.getActiveSpreadsheet(),
7         ssTarget =
8         SpreadsheetApp.create(
9             'CopySheetTest'),
10        sourceSpreadsheetName =
11        ssSource.getName(),
12        targetSpreadsheetName =
13        ssTarget.getName();
14    Logger.log(
15        'Copying the first sheet from ' +
16        sourceSpreadsheetName +
17        ' to ' + targetSpreadsheetName);
18    // [0] extracts the first Sheet object
19    // from the array created by
20    // method call "getSheets()"
21    ssSource.getSheets()[0].copyTo(ssTarget);
22 }
```

This function draws attention to some other *Spreadsheet* and *Sheet* methods. Calling it creates two *Spreadsheet* objects, one for the active spreadsheet, the second for a newly created spreadsheet. The *Spreadsheet* object names are extracted with the *getName()* method. The first sheet of the active spreadsheet is copied to the newly created spreadsheet called “CopySheetTest” that should be visible in the user’s Google Drive area. The output in the log provides a summary of the operation that was carried out.

## 4.9.4 Create A Summary Of A Sheet In A JavaScript Object Literal

The purpose of this example is to demonstrate how easy it is to interrogate a Google Apps object in order to generate a summary of its important features. For example, a user might like to know the used range in the sheet. All such information can be conveniently aggregated and labelled in a JavaScript **object literal**.

### Code Example 4.8

```
1  // Create a Sheet object and pass it
2  // as an argument to getSheetSummary().
3  // Print the return value to the log.
4  function test_getSheetSummary () {
5      var sheet = SpreadsheetApp
6          .getActiveSheet()
7          .getSheets()[0];
8      Logger.log(getSheetSummary(sheet));
9  }
10 // Given a Sheet object as an argument,
11 // use Sheet methods to extract
12 // information about it.
13 // Collect this information into an object
14 // literal and return the object literal.
15 function getSheetSummary (sheet) {
16     var sheetReport = {};
17     sheetReport['Sheet Name'] =
18         sheet.getName();
19     sheetReport['Used Row Count'] =
20         sheet.getLastRow();
21     sheetReport['Used Column count'] =
22         sheet.getLastColumn();
23     sheetReport['Used Range Address'] =
24         sheet.getDataRange().getA1Notation();
```

```
25     return sheetReport;  
26 }
```

The function `getSheetSummary()` returns a JavaScript object literal for a *Sheet* object. The calling function simply passes it a *Sheet* object argument and prints the returned object literal to the log giving this type output:

```
{Used Row Count=13, Used Range Address=A1:H13, Sheet Name=FirstSheet,  
Used Column count=8}
```

The code example 4.8 uses *Range* methods that will be discussed in the next chapter.

JavaScript object literals are extremely useful and can be used to mimic the hashes/dictionaries/maps found in other programming languages. Code example 4.8 just touches on their capabilities.

## 4.10 Summary

- Google Apps Scripts are composed of (1) Google App objects, (2) Built-in JavaScript objects, and (3) Programmer-defined JavaScript objects.
- JavaScript introspection techniques can be used to retrieve all the properties of Google Apps objects.
- The *Spreadsheet* object represents Google Spreadsheets.
- *Spreadsheet* objects are created by methods of the *SpreadsheetApp* class.
- *Spreadsheet* objects contain one or more *Sheet* objects.
- The spreadsheet cells are represented as *Range* objects (next chapter).
- *Spreadsheet*, *Sheet*, and *Range* objects can be manipulated by the methods they implement.

- JavaScript object literals are very convenient data structures for naming, collecting and returning properties of the *Spreadsheet*, *Sheet*, and *Range* objects.
- These objects and their methods will be revisited throughout the subsequent chapters.

# Chapter 5: The *Range* Object

## 5.1 Introduction

All code for this chapter is available [here on GitHub](#).<sup>24</sup>

Spreadsheet programming revolves around ranges so a good understanding of how they are manipulated programmatically by *Range* object methods is very important. Spreadsheet data is stored in one or more cells. A *Range* object represents one or more cells. *Range* objects can be created and accessed by the *Sheet* and *Spreadsheet* objects that were discussed in the previous chapter. A *Range* object can even be returned by a *SpreadsheetApp* method as shown in this function:

### Code Example 5.1

```
1 // Select a number of cells in a spreadsheet and
2 // then execute the following function.
3 // The address of the selected range, that is the
4 // active range, is written to the log.
5 function activeRangeFromSpreadsheetApp () {
6   var activeRange =
7     SpreadsheetApp.getActiveRange();
8   Logger.log(activeRange.getA1Notation());
9 }
```

The concept of an active object was discussed in the previous chapter. Range manipulation is usually performed by *Sheet* object methods. However, the *Spreadsheet* object is also required for some

---

<sup>24</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015/blob/master/cho5.gs](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/cho5.gs)

important range operations. Two that will be discussed further here are (1) the active cell and (2) range names. Each spreadsheet, not each of its constituent sheets, has an active cell and range names are unique across the spreadsheet. Range names can be assigned by users and programmers and offer many benefits that will be considered later. The active cell is a *Range* object corresponding to the selected cell, or the top-left cell of a multi-cell selection. Here is a function that determines the active cell for the active spreadsheet and prints its sheet name and address to the log:

### Code Example 5.2

```
1 // Get the active cell and print its containing
2 // sheet name and address to the log.
3 // Try re-running after adding a new sheet
4 // and selecting a cell at random.
5 function activeCellFromSheet () {
6     var activeSpreadsheet =
7         SpreadsheetApp.getActiveSpreadsheet(),
8         activeCell =
9             activeSpreadsheet.getActiveCell(),
10        activeCellSheetName =
11            activeCell.getSheet().getSheetName(),
12        activeCellAddress =
13            activeCell.getA1Notation();
14    Logger.log('The active cell is in sheet: ' +
15              activeCellSheetName);
16    Logger.log('The active cell address is: ' +
17              activeCellAddress);
18 }
```

This active cell is a *Range* object composed of a single cell. Being a *Range* object, it has methods to return its address and the sheet to which it belongs. The spreadsheet object hierarchy was discussed at length in the previous chapter but this code also shows another

facet of this hierarchy. *Sheet* objects contain *Range* objects and *Range* objects *belong* to *Sheet* objects. In the code above, the *Sheet* object containing the active cell was returned by the *Range* *getSheet()* method and the sheet name was returned by the *Sheet* *getSheetName()* method.

## 5.2 Range Objects Are Complex

Consider a single cell in any type of spreadsheet application. Each cell has a location given by its address, its containing sheet, and parent spreadsheet. It has a value and this value can be of different types such as Date, Text, or Number, for example. Its value might be a literal value or it might be returned by its formula. It may contain metadata (data about data) in the form of a comment. The cell will have a size determined by its width and height. The cell has a background colour and its contents also have a colour and a font. The cell can have a user- or programmer-defined name or it can be part of a named range. Finally, it has neighbouring cells and these neighbours all have properties. When one thinks of spreadsheet cells in this way, it is hardly a surprise to learn that their programmatic manipulation can be quite complex.

The [Google documentation](https://developers.google.com/apps-script/class_range)<sup>25</sup> lists all the *Range* object methods and their return types where defined. The JavaScript *Object.keys()* introspection technique can be used again here on the *Range* object to get a list of all its properties. Here is a function to do this:

### Code Example 5.3

---

<sup>25</sup>[https://developers.google.com/apps-script/class\\_range](https://developers.google.com/apps-script/class_range)



```
1 // Print Range object properties
2 // (all are methods) to log.
3 function printRangeMethods () {
4   var rng =
5     SpreadsheetApp.getActiveRange();
6   Logger.log(Object.keys(rng)
7     .sort().join('\n'));
8 }
```

This prints 117 methods to the log in an ascending alphabetical order. In the previous version of this book there were 108 methods. New methods for protecting and merging/de-merging ranges have been added to the latest version of Google Sheets. Many of the *Range* methods come in pairs where one member of the pair is a *get* or *set* for a single cell while the other is a *get* or *set* for a group of cells. Examples will help to highlight the more important methods, their inputs, and their return types.

## 5.3 Creating A *Range* Object

The easiest way to create a *Range* object is to call the *getRange()* method of the *Sheet* object. The complicating factor here is that the *getRange()* method has **five overloads**. Method overloading was discussed in the previous chapter. Two of the overloads for *getRange()* are shown in the following function:

### Code Example 5.4

```
1 // Creating a Range object using two different
2 // overloaded versions of the Sheet
3 // "getRange()" method.
4 // "getSheets()[0]" gets the first sheet of the
5 // array of Sheet objects returned by
6 // "getSheets()".
7 // Both calls to "getRange()" return a Range
8 // object representing the same range address
9 // (A1:B10).
10 function getRangeObject () {
11   var ss =
12     SpreadsheetApp.getActiveSpreadsheet(),
13     sh = ss.getSheets()[0],
14     rngByAddress = sh.getRange('A1:B10'),
15     rngByRowColNums =
16       sh.getRange(1, 1, 10, 2);
17   Logger.log(rngByAddress.getA1Notation());
18   Logger.log(
19     rngByRowColNums.getA1Notation());
20 }
```

The first call to *getRange()* uses what is probably the simplest overload; a single string argument denoting the range address. The second call to *getRange()* uses four integer arguments. The arguments in the order given are (1) the starting row, (2) the starting column, (3) the number of rows, and (4) the number of columns. The second call using the four integer arguments references the same range, A1:B10, as the first but is more complex.

## 5.4 Getting And Setting Range Properties

Getting and setting *Range* object properties is quite simple. The Google documentation and the GAS Script Editor are two excellent

resources for exploring this. The following function shows how to set a number of properties for a single cell:

### Code Example 5.5

```
1 // Add a new sheet.
2 // Set various properties for the cell
3 // A1 of the new sheet.
4 function setRangeA1Properties() {
5     var ss =
6         SpreadsheetApp.getActiveSpreadsheet(),
7         newSheet,
8         rngA1;
9     newSheet = ss.insertSheet();
10    newSheet.setName('RangeTest');
11    rngA1 = newSheet.getRange('A1');
12    rngA1.setNote(
13        'Hold The date returned by spreadsheet '
14        + ' function "TODAY()"');
15    rngA1.setFormula('=TODAY()');
16    rngA1.setBackground('black');
17    rngA1.setFontColor('white');
18    rngA1.setFontWeight('bold');
19 }
```

This function first adds a new sheet to the active spreadsheet. Then a number of set methods of the *Range* object are called to set some visible properties of its cell A1. The outcome can be seen in the figure 5-1 below.

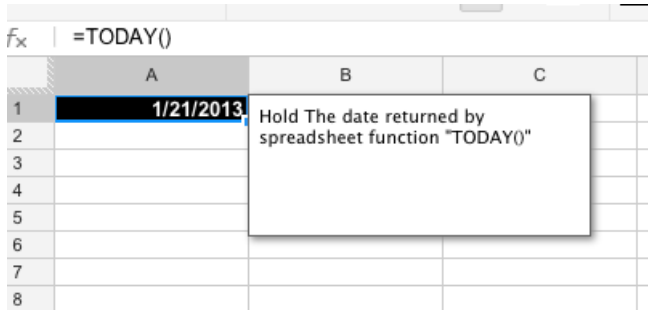


Figure 5-1: Cell A1 after the function `setRangeA1Properties()` has been executed showing the cell formula, value, comment, and background and font colours.

The following function demonstrates *get* methods of the *Range* object. When executed it writes the values of the properties set above to the log:

#### Code Example 5.6

```

1  // Demonstrate get methods for 'Range'
2  // properties.
3  // Assumes function "setRangeA1Properties()"
4  // has been run.
5  // Prints the properties to the log.
6  // Demo purposes only!
7  function printA1PropertiesToLog () {
8      var rngA1 =
9          SpreadsheetApp.getActiveSpreadsheet()
10         .getSheetByName('RangeTest').getRange('A1');
11     Logger.log(rngA1.getNote());
12     Logger.log(rngA1.getFormula());
13     Logger.log(rngA1.getBackground());
14     Logger.log(rngA1.getFontColor());
15     Logger.log(rngA1.getFontWeight());
16 }

```

The output in the logger shows the values that were set for the range

properties in the previous function.

These two functions show how easy it is to alter and retrieve the properties of the *Range* object. A good way to learn more is to experiment with these code examples and see what happens. In this example the *Range* object represented just a single cell but the methods used can just as easily be applied to blocks of cells, an entire column or row, or even an entire sheet.

## 5.5 The *Range* *offset()* Method

Every cell in a spreadsheet has neighbouring cells. These neighbouring cells can be referenced by the very useful *Range* object *offset()* method. The method *offset()* is overloaded (see the overloading discussion in the previous chapter). In its simplest form, it requires two integer arguments; the row offset and the column offset. The following function starts with the cell C10 of the active sheet and adds comments to all its adjoining cells describing what row and column offsets were used:

### Code Example 5.7

```
1 // Starting with cell C10 of the active sheet,
2 // add comments to each of its adjoining cells
3 // stating the row and column offsets needed
4 // to reference the commented cell
5 // from cell C10.
6 function rangeOffsetDemo () {
7     var rng =
8         SpreadsheetApp.getActiveSheet()
9             .getRange('C10');
10    rng.setBackground('red');
11    rng.setValue('Method offset()');
12    rng.offset(-1,-1)
13        .setNote('Offset -1, -1 from cell ')
```

```
14     + rng.getA1Notation());
15   rng.offset(-1,0)
16     .setNote('Offset -1, 0 from cell '
17     + rng.getA1Notation());
18   rng.offset(-1,1)
19     .setNote('Offset -1, 1 from cell '
20     + rng.getA1Notation());
21   rng.offset(0,1)
22     .setNote('Offset 0, 1 from cell '
23     + rng.getA1Notation());
24   rng.offset(1,0)
25     .setNote('Offset 1, 0 from cell '
26     + rng.getA1Notation());
27   rng.offset(0,1)
28     .setNote('Offset 0, 1 from cell '
29     + rng.getA1Notation());
30   rng.offset(1,1)
31     .setNote('Offset 1, 1 from cell '
32     + rng.getA1Notation());
33   rng.offset(0,-1)
34     .setNote('Offset 0, -1 from cell '
35     + rng.getA1Notation());
36   rng.offset(1,-1)
37     .setNote('Offset -1, -1 from cell '
38     + rng.getA1Notation());
39 }
```

The effect of this function can be seen in figure 5-2 below:

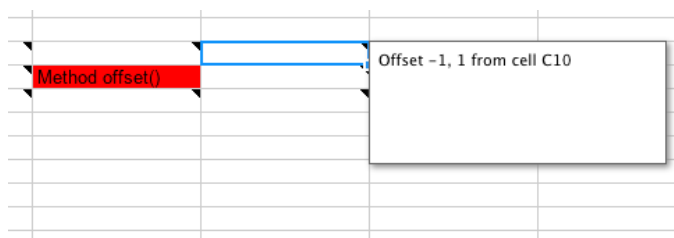


Figure 5-2: Comments added to cells denoting the *offset()* arguments required to reference them from the cell shown in red.

The offset values given can, of course, have a magnitude greater than one.

## 5.6 The Sheet Data Range

It is important to know the range of cells in a sheet bounded by and including the bottom-most populated row and left-most populated column. This range comprises what Excel refers to as the **used range**. In Google Apps Script, it is referred to as the **data range** and it can be captured as a *Range* object using the *Sheet getDataRange()* method. The Google documentation gives a succinct definition of the range this method returns: “Returns a *Range* corresponding to the dimensions in which data is present.”

To demonstrate the data range, a small data set showing a snapshot of the points ranking of the English football Premier League was copied into a sheet named “english\_premier\_league”, see Figure 5-3. The source data can be found in a shared, read-only spreadsheet [here](https://docs.google.com/spreadsheets/d/1pBXF2VY75rjBZY5uM--X1z-e2E7Pq_QWIpxKexF7Afc/edit?usp=sharing)<sup>26</sup>. This data set will be used in other examples that follow in this chapter. The data itself is just three columns, a header in row 1 and 20 data rows with one row for each club in the English football Premier League. The first column contains the club name,

<sup>26</sup>[https://docs.google.com/spreadsheets/d/1pBXF2VY75rjBZY5uM--X1z-e2E7Pq\\_QWIpxKexF7Afc/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1pBXF2VY75rjBZY5uM--X1z-e2E7Pq_QWIpxKexF7Afc/edit?usp=sharing)

the second is the number of games played, and the third is the points total at the time the data was copied:

	A	B	C	D
1	<b>Club</b>	<b>Played</b>	<b>Pts</b>	
2	Chelsea	32	76	
3	Arsenal	32	66	
4	Liverpool	32	57	
5	Sunderland	32	29	
6	Hull City	32	28	
7	Leicester City	32	28	
8	Man United	33	65	
9	Man City	33	64	
10	Tottenham	33	57	
11	Southampton	33	56	
12	Swansea City	33	47	
13	Stoke City	33	46	
14	West Ham	33	43	
15	Crystal Palace	33	42	
16	Everton	33	41	
17	West Brom	33	36	
18	Newcastle	33	35	
19	Aston Villa	33	32	
20	QPR	33	26	
21	Burnley FC	33	26	

Figure 5-3: Sample data to demonstrate the data range of a sheet.

To see how the data range can be determined, execute the following code from the Script Editor:

### Code Example 5.8



```
1 // See the Sheet method getDataRange() in action.
2 // Print the range address of the used range for
3 // a sheet to the log.
4 // Generates an error if the sheet name
5 // "english_premier_league" does not exist.
6 function getDataRange () {
7     var ss =
8         SpreadsheetApp.getActiveSpreadsheet(),
9         sheetName = 'english_premier_league',
10        sh = ss.getSheetByName(sheetName),
11        dataRange = sh.getDataRange();
12    Logger.log(dataRange.getA1Notation());
13 }
```

This prints “A1:C21”. To see the dynamic nature of the data range, add some content to cell D23. The code above now prints “A1:D23”. Note that, even though cell D23 is surrounded by blank cells, adding content to it causes the data range to expand to include its column and all columns to its left as well as its row and all rows above it. This means that as data is added, the data range can be re-calculated to include the new content. If cell contents are deleted, the data range adjusts accordingly. However, cell comments and cell formats do not affect the data range. Adding a cell comment alone to cell D23 will not alter the data range. Similarly, adding formatting to a cell does not either. Only adding or deleting cell content affects the data range.

## 5.7 Transferring Values Between JavaScript Arrays And Ranges

This section assumes a basic understanding of JavaScript arrays. If required, consult the resources listed in Appendix B to learn about them.

Once a *Range* object has been created all the values of its cells can be read into a JavaScript array with one method call; the *Range* object *getValues()* method. Using the data given in the previous example, here is a function that reads all of it into a JavaScript array.

### Code Example 5.9

```
1 // Read the entire data range of a sheet
2 // into a JavaScript array.
3 // Uses the JavaScript Array.isArray()
4 // method twice to verify that method
5 // getValues() returns an array-of-arrays.
6 // Print the number of array elements
7 // corresponding to the number of data
8 // range rows.
9 // Extract and print the first 10
10 // elements of the array using the
11 // array slice() method.
12 function dataRangeToArray () {
13     var ss =
14         SpreadsheetApp.getActiveSpreadsheet(),
15         sheetName = 'english_premier_league',
16         sh = ss.getSheetByName(sheetName),
17         dataRange = sh.getDataRange(),
18         dataRangeValues = dataRange.getValues();
19     Logger.log(Array.isArray(dataRangeValues));
20     Logger.log(Array.isArray(dataRangeValues[0]));
21     Logger.log(dataRangeValues.length);
22     Logger.log(dataRangeValues.slice(0, 10));
23 }
```

The *Range* object *getValues()* returns a JavaScript array-of-arrays. This is confirmed by the JavaScript *Array.isArray()* method where the check is run on the full array and on its first element. Even if the input range has only a single column, the *getValues()* method

will still return an array-of-arrays where the inner arrays each will have just a single element.

Once the data range values have been loaded into a JavaScript array, they are amenable to JavaScript's powerful array methods. GAS supports all the modern JavaScript array methods so when reviewing JavaScript arrays, make sure to check up-to-date sources. Excellent JavaScript array documentation is available on [this site](#).<sup>27</sup>

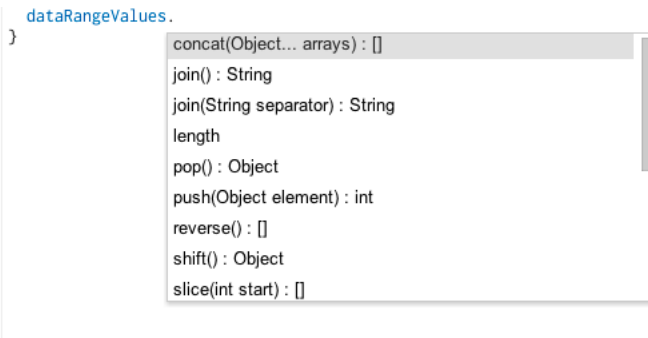


Figure 5-4: The Script Editor showing some of the JavaScript array methods available in GAS.

The array *slice()* method, for example, is used above to extract the first 10 elements of the array. The first integer argument specifies the array index to start the slice. When called without a second argument, the *slice()* method will return the rest of the array. When the second argument is given, the slice operation stops at the index up to, but not including, that value.

The easiest way to loop through the array and to process its elements is with a JavaScript *for* loop as shown in the following code example.

### Code Example 5.10

<sup>27</sup>[https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array)

```
1 // Loop over the array returned by
2 // getRange() and a CSV-type output
3 // to the log using array join() method.
4 function loopOverArray () {
5     var ss =
6         SpreadsheetApp.getActiveSpreadsheet(),
7         sheetName = 'english_premier_league',
8         sh = ss.getSheetByName(sheetName),
9         dataRange = sh.getDataRange(),
10        dataRangeValues =
11            dataRange.getValues(),
12        i;
13    for ( i = 0;
14          i < dataRangeValues.length;
15          i += 1 ) {
16        Logger.log(
17            dataRangeValues[i].join(', '));
18    }
19 }
```

The output in the logger resembles Comma-Separated Value (CSV) output. Each iteration of the *for* loop fetches one of the inner arrays and calls the array *join()* method to create a string from the array with elements separated by a comma (“,”).

The next code example is much more complicated and performs more tasks than one would normally expect from a well-written function. It shows how to manipulate the array returned from the *Range* *getValues()* method using JavaScript array methods. Its purpose is to filter the input array to populate a new array with only those elements that meet a filter criterion, that is, retrieve only those elements where the points total is 40 or more.

### Code Example 5.11

```
1 // In production code, this function would be
2 // re-factored into smaller functions.
3 // Read the data range into a JavaScript array.
4 // Remove and store the header line using the
5 // array shift() method.
6 // Use the array filter() method with an anonymous
7 // function as a callback to implement the
8 // filtering logic.
9 // Determine the element count of the
10 // filter() output array.
11 // Add a new sheet and store a reference to it.
12 // Create a Range object from the new
13 // Sheet object using the getRange() method.
14 // The four arguments given to getRange() are:
15 // (1) first column, (2) first row,
16 // (3) last row, and (4) last column.
17 // This creates a range corresponding to
18 // range address "A1:C5".
19 // Write the values of the filtered array to the
20 // newly created sheet.
21 function writeFilteredArrayToRange () {
22     var ss =
23         SpreadsheetApp.getActiveSpreadsheet(),
24         sheetName = 'english_premier_league',
25         sh = ss.getSheetByName(sheetName),
26         dataRange = sh.getDataRange(),
27         dataRangeValues = dataRange.getValues(),
28         filteredArray,
29         header = dataRangeValues.shift(),
30         filteredArray,
31         filteredArrayColCount = 3,
32         filteredArrayRowCount,
33         newSheet,
34         outputRange;
35     filteredArray = dataRangeValues.filter(
```

```
36     function (innerArray) {
37         if (innerArray[2] >= 40) {
38             return innerArray;
39         }
40     filteredArray.unshift(header);
41     filteredArrayRowCount = filteredArray.length;
42     newSheet = ss.insertSheet();
43     outputRange = newSheet
44         .getRange(1,
45                 1,
46                 filteredArrayRowCount,
47                 filteredArrayColCount);
48     outputRange.setValues(filteredArray);
49 }
```

After the code above has executed, a new sheet will be visible with with a row for each . Because this function is more complex than other examples and introduces more advanced JavaScript techniques than used previously, a more detailed explanation of its operations is given below.

This function:

- Reads the data range from the data sheet into a JavaScript array-of-arrays as described previously.
- Removes the first element that represents the header row of the data set using the JavaScript array *shift()* method and stores it in a variable.
- Populates a new array using the JavaScript array *filter()* method by passing it an anonymous function as a **callback**.
- Uses the callback to run the filter test returning only elements where the third element (points) is greater than or equal to 40.
- Adds the header row back to the filtered array using the JavaScript array *unshift()* method.

- Determines the number of elements in the filtered array.
- Uses the four-argument overload of the *Range* object *setValues()* method to copy the filtered array to the newly inserted sheet.

The array element corresponding to the data header row is removed and stored because the subsequent filtering uses a numerical expression that is invalid for the text values it contains. The JavaScript *shift()* array method that does this is one of the JavaScript array **mutator** methods. These mutator methods are so-called because they alter the array object that calls them. Other mutator JavaScript array methods are *splice()*, *pop()*, *push()* and *unshift()*.

As stated in the user-defined chapter, JavaScript functions are extremely powerful. The JavaScript array *filter()* method demonstrates this. It takes an **anonymous function** as an argument and uses this function as a so-called **callback** to filter the array. Other JavaScript array methods that take callbacks as arguments are *map()* and *forEach()*. The Mozilla link given earlier describes all these JavaScript array methods very clearly.

In order to use the *Range* object *setValues()* method with the array-of-arrays as its single argument, the dimensions of the target *Range* object (that is, its row and column count) have to match exactly the dimensions of the array. The array “column” count is the length of the inner arrays and that is known to be 3 and will not be altered by the filtering. The row count, that is the length of the outer array is determined after the filtering and re-insertion of the header row.

The *filter()* method and its callback argument could be replaced by a straight-forward JavaScript *for* loop. However, callbacks are one of JavaScript’s best features and are well worth exploring.

## 5.8 Named Ranges

Assigning names to ranges is a useful spreadsheet technique. Range names are unique across the spreadsheet while addresses are not. Every sheet within a spreadsheet contains a cell with a range address *A1*. Named ranges can also be protected to avoid inadvertent changes to important values. When a range name is re-assigned to a different range, its previous association is lost and no warning is given. Excel also exhibits this behaviour and, as later examples will show, it can actually be quite useful.

Named ranges are defined in Google Spreadsheets using the menu sequence “*Data->Named ranges..*” and clicking the “*Add a range*” in the resulting dialog. Enter a name for the range and the range address in the dialog see Figure 5-5. The mouse right-click also brings up this display.

Named ranges simplify lookup functions such as *VLOOKUP* where the range name can be used in place of the “\$” signs to create an absolute reference to the input. Incorrect usage of these \$ characters and omission of the final “*FALSE*” argument are common sources of *VLOOKUP* errors, especially when the formula is copied down using the bottom right “*black cross*”. Using a range name eliminates the need for the \$ characters and makes the *VLOOKUP* much easier to read and debug.

Take the English football premier league table as an example. When the data range is given a range name “*league\_table*” the built-in *VLOOKUP* spreadsheet function can be used as follows in any cell in the spreadsheet:

```
1 =VLOOKUP("Arsenal", league_table, 3, FALSE)
```

This *VLOOKUP* reports the points total for Arsenal.

If an error of this type “*error: Did not find value*” is reported in the



*VLOOKUP* cell comment section then the test value is considered to be absent.



## Watch Out For White Space.

The sample data copied from the indicated website had trailing whitespace. Leading and trailing white space can cause lookup functions to fail.

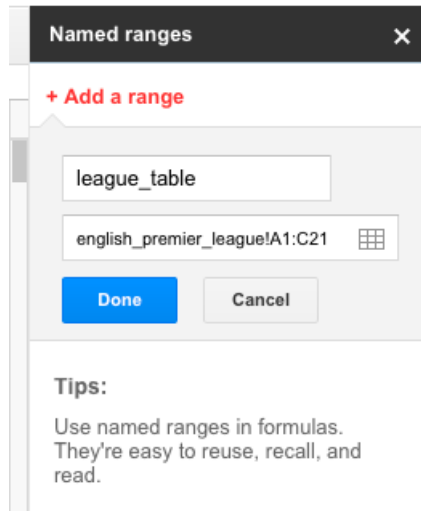


Figure 5-5: Dialog for creating a named range in Google Spreadsheets.

Range names adjust to accommodate row/column insert/delete operations within their boundaries. They also move as necessary to accommodate insert/delete operations outside their boundaries. This stability is one of their most advantageous features.

Because range names are unique across the spreadsheet, the methods used for them belong to the *Spreadsheet* object. The range name methods with their arguments are:

1. `setNamedRange(name, range)`

2. `removeNamedRange(name)`
3. `getRangeByName (name)`

This code sets a range name in the active sheet of the active spreadsheet:

### Code Example 5.12

```
1 // Add a name to a range.
2 function setRangeName () {
3   var ss =
4     SpreadsheetApp.getActiveSpreadsheet(),
5     sh = ss.getActiveSheet(),
6     rng = sh.getRange('A1:B10'),
7     rngName = 'MyData';
8   ss.setNamedRange(rngName, rng);
9 }
```

Once the range name is set, it can be used to create *Range* objects representing the named range. Annoyingly, there is no way to get an array of all the named ranges in a spreadsheet as one can do in Excel. The issue has been raised but, as of April 2015, it has not yet been addressed by Google, see [this link](#)<sup>28</sup>.

## 5.9 Practical Examples

The following examples aim to clarify some of topics covered earlier in this chapter, to introduce some new *Range* methods, and to provide some useful examples that readers can copy and adapt to their own requirements. The examples usually come as a pair of functions where the one prefixed with “call\_” and written in boldface is to be executed.

---

<sup>28</sup><https://code.google.com/p/google-apps-script-issues/issues/detail?id=917>

## 5.9.1 Change The Colour Of Every Second Row

Spreadsheet users frequently request tips on how to add a colour to every second row of a spreadsheet range. This type of effect is often used with HTML tables using Cascading Style Sheets (CSS). The same effect is quite easy to achieve using GAS and is code to this is given in example 5.13 below.

### Code Example 5.13

```
1 // Create a range object using the
2 // getDataRange() method.
3 // Pass the range and a colour name
4 // to function "setAlternateRowsColor()".
5 // Try changing the 'color' variable to
6 // something like:
7 // 'red', 'green', 'yellow', 'gray', etc.
8 function call_setAlternateRowsColor () {
9     var ss =
10         SpreadsheetApp.getActiveSpreadsheet(),
11         sheetName = 'english_premier_league',
12         sh = ss.getSheetByName(sheetName),
13         dataRange = sh.getDataRange(),
14         color = 'grey';
15     setAlternateRowsColor(dataRange, color);
16 }
17
18 // Set every second row in a given range to
19 // the given colour.
20 // Check for two arguments:
21 // 1: Range, 2: string for colour.
22 // Throw a type error if either argument
23 // is missing or of the wrong type.
24 // Use the Range offset() to loop
25 // over the range rows.
```

```
26 // the for loop counter starts at 0.
27 // It is tested in each iteration with the
28 // modulus operator (%).
29 // If i is an odd number, the if condition
30 // evaluates to true and the colour
31 // change is applied.
32 // WARNING: If a non-existent colour is given,
33 // then the "color" is set to undefined
34 // (no color). NO error is thrown!
35 function setAlternateRowsColor (range,
36                                 color) {
37     var i,
38         startCell = range.getCell(1,1),
39         columnCount = range.getLastColumn(),
40         lastRow = range.getLastRow();
41     for (i = 0; i < lastRow; i += 1) {
42         if (i % 2) {
43             startCell.offset(i, 0, 1, columnCount)
44                 .setBackgroundColor(color);
45         }
46     }
47 }
```

The visual effect of the above code can be see in figure 5-6 below.

	A	B	C
1	<b>Club</b>	<b>Played</b>	<b>Pts</b>
2	Chelsea	32	76
3	Arsenal	32	66
4	Liverpool	32	57
5	Sunderland	32	29
6	Hull City	32	28
7	Leicester City	32	28
8	Man United	33	65
9	Man City	33	64
10	Tottenham	33	57
11	Southampton	33	56
12	Swansea City	33	47
13	Stoke City	33	46
14	West Ham	33	43
15	Crystal Palace	33	42
16	Everton	33	41
17	West Brom	33	36
18	Newcastle	33	35
19	Aston Villa	33	32
20	QPR	33	26
21	Burnley FC	33	26

Figure 5-6: Spreadsheet appearance after setting the background colour of alternate rows in the data range to “gray”.

Note: Google Apps Script uses American spelling, hence “color” but the text here uses the UK “colour”. Note also that the colour spelled “grey” in the UK and elsewhere is spelled “gray” in the US. Google Apps Script recognises both spellings.

The function *setAlternateRowsColor()* takes two arguments: A *Range* object and a string for the colour. The familiar argument checking is performed to ensure that two arguments of the right type have been passed. The *Range* object *offset()* method is then executed in a *for* loop. The four argument overload of *offset()* is used where the arguments are (1) the row offset (the *i* loop counter variable), (2) the column offset, (3) the number of rows, and (4) the number

of columns. The effect of this call to the *offset()* method is to select one row with three columns in each loop iteration.

The **modulus (%)** operator is used with the loop variable (*i*) and if it is odd, then the *if* condition evaluates to *true* and the colour of the row is changed.



### Unrecognised Colour Names.

When a non-existent colour name is passed to *setAlternateRowsColor()*, any previous row colour is removed!

Try running the code to set the row colour to some legitimate colour like 'red', then re-run it with a non-existent colour name like 'grime' and see the red disappear. One might have expected an error in these situations but that does not happen!

## 5.9.2 Remove Leading And Trailing Whitespace From A *Range*

Leading and trailing spaces in cell values are difficult to see, that is especially true of the trailing type. They can cause problems with spreadsheet lookup functions such as *VLOOKUP*. Google Sheets supports the text function called *TRIM()* which removes them but sometimes it is preferable to run this operation programmatically. The following function called *deleteLeadingTrailingSpaces()* takes a *Range* object as an argument and loops through each cell in the range and, if the cell contains text, it uses the JavaScript *String* object *trim()* method to remove leading and trailing spaces.

**Code Example 5.14**

```
1 // Test function for
2 // "deleteLeadingTrailingSpaces()".
3 // Creates a Range object and passes
4 // it to this function.
5 function call_deleteLeadingTrailingSpaces() {
6     var ss =
7         SpreadsheetApp.getActiveSpreadsheet(),
8         sheetName = 'english_premier_league',
9         sh = ss.getSheetByName(sheetName),
10        dataRange = sh.getDataRange();
11    deleteLeadingTrailingSpaces(dataRange);
12 }
13
14 // Process each cell in the given range.
15 // If the cell is of type text
16 // (typeof === 'string') then
17 // remove leading and trailing white space.
18 // Else ignore it.
19 // Code note: The Range getCell() method
20 // takes two 1-based indexes
21 // (row and column).
22 // This is in contrast to the offset() method.
23 // rng.getCell(0,0) will throw an error!
24 function deleteLeadingTrailingSpaces(range) {
25     var i,
26         j,
27         lastCol = range.getLastColumn(),
28         lastRow = range.getLastRow(),
29         cell,
30         cellValue;
31     for (i = 1; i <= lastRow; i += 1) {
32         for (j = 1; j <= lastCol; j += 1) {
33             cell = range.getCell(i,j);
34             cellValue = cell.getValue();
35             if (typeof cellValue === 'string') {
```

```
36         cellValue = cellValue.trim();
37         cell.setValue(cellValue);
38     }
39 }
40 }
41 }
```

The function *deleteLeadingTrailingSpaces()* demonstrates how to loop through a range one cell at a time using the *Range getCell()* method with the outer and inner *for* loop variables, *i* and *j*, providing the row and column indexes, respectively. While the code in *deleteLeadingTrailingSpaces()* works as expected, it is not the most efficient way of processing a range of cells in GAS. A better approach in GAS, and one that will be used in later examples, is to transfer the entire set of values for the range into a nested array in one go using the *Range getValues()* method, process the nested array and then write the new values back to the original range using the *Range setValues()* method.



## Zero- Or One-Based Indexes

It is important to be aware that the *getCell()* method uses one-based index so that the first cell in a range is *rng\_object.getCell(1, 1)*. The *offset()* method, in contrast, is zero-based.

Using a callback could conveniently separate the tasks of looping through a range of cells one-by-one and trimming cells with text. In this manner different callbacks could be applied to different tasks. One callback function could do trimming as shown above while another could do something else.



### 5.9.3 Extract Details Of All Formulas In a Sheet

One of the defining characteristics of spreadsheet applications is the ability to write formulas in cells. Using named ranges and cell comments is very helpful for documenting complex spreadsheet applications where there are multiple cells with formulas spread across different sheets. The following example code example contains two functions. The main one, entitled “*getAllDataRangeFormulas()*” checks the entire data range (used range) of a *Sheet* object argument and returns an object literal containing the details of each formula found. Figure 5-6 below shows the data in a test sheet.

	A	B	C	D	E
1	"=TODAY()"	1/28/2013	5	Formula: =INT(RAND()*6 + 1)	
2	"=NOW()"	1/28/2013 19:56:17	3		
3			6		
4			4		
5			2		
6			1		
7			2		
8			5		
9			28		
10					
11					

Figure 5-7: A sheet with some formulas to test the function *getAllDataRangeFormulas()*.

#### Code Example 5.15

```

1 // Create a Sheet object for the active sheet.
2 // Pass the sheet object to
3 // "getAllDataRangeFormulas()"
4 // Create an array of the keys in the returned
5 // object in default "sort()".
6 // Loop over the array of sorted keys and
7 // extract the values they keys map to.
8 // Write the output to the log.
9 function call_getAllDataRangeFormulas() {
10   var ss =
11     SpreadsheetApp.getActiveSpreadsheet(),
12     sheet = ss.getActiveSheet(),

```

```
13     sheetFormulas = getAllDataRangeFormulas(sheet),
14     formulaLocations =
15         Object.keys(sheetFormulas).sort(),
16     formulaCount = formulaLocations.length,
17     i;
18     for (i = 0; i < formulaCount; i += 1) {
19         Logger.log(formulaLocations[i] +
20             ' contains ' +
21                 sheetFormulas[formulaLocations[i]]);
22     }
23 }
24 // Take a Sheet object as an argument.
25 // Return an object literal where formula
26 // locations map to formulas for all formulas
27 // in the input sheet data range.
28 // Loop through every cell in the data range.
29 // If a cell has a formula,
30 // store that cells location as
31 // the key and its formula as the value
32 // in the object literal.
33 // Return the populated object literal.
34 function getAllDataRangeFormulas(sheet) {
35     var dataRange = sheet.getDataRange(),
36         i,
37         j,
38         lastCol = dataRange.getLastColumn(),
39         lastRow = dataRange.getLastRow(),
40         cell,
41         cellFormula,
42         formulasLocations = {},
43         sheetName = sheet.getSheetName(),
44         cellAddress;
45     for (i = 1; i <= lastRow; i += 1) {
46         for (j = 1; j <= lastCol; j += 1) {
47             cell = dataRange.getCell(i,j);
```

```

48     cellFormula = cell.getFormula();
49     if (cellFormula) {
50         cellAddress = sheetName + '!' +
51             cell.getA1Notation();
52         formulasLocations[cellAddress] =
53             cellFormula;
54     }
55 }
56 }
57 return formulasLocations;
58 }

```

After calling “*call\_getAllDataRangeFormulas()*”, details for all the formulas in the test sheet will appear in the log. Here are the first few lines of output from the sheet shown in figure 4-6:

Sheet2!B1 contains =today() Sheet2!B2 contains =NOW() Sheet2!C1 contains =INT(RAND()\*6 + 1) Etc .....

The working function here, like the earlier leading/trailing spaces example, loops through all cells in an input range. However, it does not modify any cells. Instead of modifications, it uses some additional *Range* methods to extract information about each cell. The two *Range* methods it uses to perform its task are (1) *getFormula()* and (2) *getA1Notation()*. The names are self-explanatory and the values they extract are then used to populate the object literal that stores details for all cells in the input sheet that contain a formula.

### 5.9.4 Copy Columns From One Sheet To Another In A Specified Order

The columns in a spreadsheet sheet may need to be re-ordered or, alternatively, only a sub-set of the columns may be required. Relational databases support SQL and the SQL *SELECT* statement can be used to do this. For Google Spreadsheets, this functionality can be implemented quite easily using GAS.

The function `copyColumns()` in the following code example can be used to:

1. Select of sub-set of sheet columns.
2. Re-order the sheet columns.

### Code Example 5.16

```
1 // Call copyColumns() function passing it:
2 // 1: The active sheet
3 // 2: A newly inserted sheet
4 // 3: An array of column indexes to copy
5 //    to the new sheet
6 // The output in the newly inserted sheet
7 // contains the columns for the indexes
8 // given in the array in the
9 // order specified in the array.
10 function run_copyColumns() {
11     var ss =
12         SpreadsheetApp.getActiveSpreadsheet(),
13         inputSheet = ss.getActiveSheet(),
14         outputSheet = ss.insertSheet(),
15         columnIndex = [4,3,2,1];
16     copyColumns(inputSheet,
17                 outputSheet,
18                 columnIndex);
19 }
20 // Given an input sheet, an output sheet,
21 // and an array:
22 // Use the numeric column indexes in
23 // the array to copy those columns from
24 // the input sheet to the output sheet.
25 // The function checks its input arguments
26 // and throws an error
```

```
27 // if they are not Sheet, Sheet, Array.
28 // The array is expected to be an array of
29 // integers but it does
30 // not check the array element types
31 function copyColumns(inputSheet,
32                     outputSheet,
33                     columnIndexes) {
34   var dataRangeRowCount =
35     inputSheet.getDataRange()
36       .getNumRows(),
37     columnsToCopyCount =
38       columnIndexes.length,
39     i,
40     columnIndexesCount,
41     valuesToCopy = [];
42   for (i = 0;
43        i < columnsToCopyCount;
44        i += 1) {
45     valuesToCopy =
46       inputSheet
47         .getRange(1,
48                  columnIndexes[i],
49                  dataRangeRowCount,
50                  1).getValues();
51     outputSheet
52       .getRange(1,
53                i+1,
54                dataRangeRowCount,
55                1).setValues(valuesToCopy);
56   }
57 }
```

I have created some sample data for testing code example 5.16; It is

in [this shared, read-only Google Sheet](#)<sup>29</sup>. To run, copy the data from this sheet into your own Google Sheet, copy code example 5.16 into the Script Editor and execute function `run_copyColumns()`. Once this function has executed, you should see new sheet that looks like figure 5-8 below.

The `copyColumns()` function uses the *Range* object represented by the data range to determine the number of rows to copy. It processes each column index in the *for* loop. It uses the four-argument overload of the *Sheet* `getRange()` method to define the target range and the *Range* object `getValues()` method to extract the cell values into a JavaScript array-of-arrays where the inner arrays each contain a single element.

The JavaScript array is then written to the output sheet using the *Range* object `setValues()` method. This method expects the dimensions of the target range to exactly match those of its JavaScript array argument.

One point of note here is that the *Sheet* `getRange()` method uses one-based indexes but JavaScript array indexes are zero-based. This explains the  $i+1$  argument in the `getRange()` method call. See the earlier warning about `offset()` and `getRange()` indexes!

---

<sup>29</sup><https://docs.google.com/spreadsheets/d/1E5qAdwmKh2KSGYv1G4MwtSFMhlnGvyCpxETxKnRqNjM/edit?usp=sharing>

	A	B	C	D
1	W	GP	Team	Position
2	23	32	Chelsea	1
3	20	32	Arsenal	2
4	19	33	Man United	3
5	19	33	Man City	4
6	17	32	Liverpool	5
7	17	33	Tottenham	6
8	17	33	Southampton	7
9	13	33	Swansea City	8
10	13	33	Stoke City	9
11	11	33	West Ham	10
12	11	33	Crystal Palace	11
13	10	33	Everton	12
14	9	33	West Brom	13
15	9	33	Newcastle	14
16	8	33	Aston Villa	15
17	5	32	Sunderland	16
18	6	32	Hull City	17
19	7	32	Leicester City	18
20	7	33	QPR	19
21	5	33	Burnley FC	20

Figure 5-8: Sub-set column output produced by the copyColumn function.

## 5.11 Concluding Remarks

This and the previous chapter have covered the fundamental objects, *Spreadsheet*, *Sheet*, and *Range*, required for Google Spreadsheet programming. It would not be practical to go through every method exposed by these objects but these chapters should provide an adequate basis for getting started. Furthermore, each of these objects will be discussed further, when the need arises, in later chapters. However, before proceeding beyond this point, it is important to have a good grasp of the material covered so far.

To achieve this, you should run the code examples as given and then experiment with them. Also, try all the examples given in the Google documentation.

## 5.12 Summary

- *Range* objects are generally created, accessed and manipulated through *Sheet* objects.
- Range names, the active range, and the active cell are accessed through the *Spreadsheet* object.
- The *Spreadsheet*, *Sheet*, and *Range* objects contain a lot of methods.
- Many *Spreadsheet*, *Sheet*, and *Range* methods are overloaded.
- JavaScript introspection techniques, the Google Apps Script Editor, and Google documentation are all very useful for identifying methods and determining what they do, their arguments, and return values.
- Named ranges simplify many spreadsheet tasks such as lookups with *VLOOKUP*.
- *Range* methods make it easy to transfer spreadsheet range values to and from JavaScript arrays.
- Important *Range* methods covered in this chapter include *offset()*, *getValues()*, *setValues()*, and *getCell()*.
- JavaScript arrays offer powerful methods for data manipulation.



# Chapter 6: MySQL And JDBC

## 6.1 Introduction

Spreadsheet applications can be used effectively as simple databases when the data volume is small and the number of users accessing the spreadsheet is low. Many users start out using a spreadsheet as a database but then migrate their data to a database application as data volume and data usage increases.

This chapter shows how Google Sheets can interact with relational databases via the Google *JDBC* API. Large data sets that are stored securely in the cloud can be manipulated in Google Sheets thereby combining the best of both spreadsheet and relational database approaches. The relational database is a mature technology dating back to the early 1970s and addresses concerns around data security, data integrity, and data scalability.

The application that manages relational databases is called the Relational Database Management System (RDBMS). Structured Query Language (SQL) is the *lingua franca* of RDBMSs where it is used not just for querying stored data, but also for creating key database objects such as tables, views and indexes. In addition, it is used to define database constraints and to manage users and permissions. As will be seen in the examples that follow in this chapter, the *JDBC* API uses SQL to interact with the database so basic SQL knowledge, or a willingness to learn SQL, is assumed. Each RDBMS has its own SQL “dialect” and some changes, generally minor, are required in order to get SQL written for one RDMS to work in another.

MySQL was chosen as the RDBMS for the examples here because it is widely used, it is available for free, and cloud test instances can be set up for minimal or no cost. Although MySQL is not as feature-

rich as either Oracle or SQL Server, it is adequate for the purposes described in this chapter. Two other excellent open source and free RDBMSs are [SQLite](#)<sup>30</sup> and [PostgreSQL](#)<sup>31</sup> but at the time of writing neither was available for GAS.

Google offer a cloud-hosted MySQL called [Google Cloud SQL](#)<sup>32</sup>. I have used this service in the previous version of this chapter but it is not free so this time I have chosen a different provider that does offer a free version. The one I have chosen is [Free MySQL Hosting](#).<sup>33</sup> I actually pay \$14 a year to this provider to get some extra space but there is a free version for trying it out. This was the first free provider I found and my choice here is not an endorsement. Depending on your requirements, you might feel it necessary to go with Google's offering. Eventualities such as when a company suspends or cancels a service or goes out of business entirely should be considered when selecting a cloud data store; Google might charge a small fee but there is a strong guarantee that it will still be operating for years to come.

Regardless of the vendor you choose, the same principles apply when establishing a GAS connection to the database and running queries against it. To perform these tasks, you use the *JDBC* API.

## 6.2 What Is *JDBC*?

The *JDBC* API was developed for the Java language for accessing databases, for a general introduction, see the [Wikipedia entry](#)<sup>34</sup>. Google have ported the *JDBC* API so that it can be used within GAS thereby providing a mechanism to allow Google Sheets to interact with relational databases. There is plenty of *JDBC* information and

---

<sup>30</sup><http://www.sqlite.org/>

<sup>31</sup><http://www.postgresql.org/>

<sup>32</sup><https://cloud.google.com/sql/docs/introduction>

<sup>33</sup><http://www.freemysqlhosting.net/>

<sup>34</sup>[http://en.wikipedia.org/wiki/Java\\_Database\\_Connectivity](http://en.wikipedia.org/wiki/Java_Database_Connectivity)

documentation with examples in Java that can be re-written in GAS without much effort.

Those who have used ActiveX Data Objects (ADO) in Microsoft Excel VBA to query and manipulate relational databases will feel quite comfortable with the *JDBC* API. The basic approach of both APIs is similar in that the programmer establishes a database connection, issues SQL queries against the database, processes record sets, uses bind variables in prepared statements and so on.

The *JDBC* API uses SQL to perform actions on the database. These include so called **CRUD** actions (Create, Read, Update, and Delete). The code examples given here should require minimal modification for use with either Oracle or SQL Server though none of the code has been tested against these products. There will be differences in how the database connections are created and in how databases are created. Otherwise, the code should be largely “database-agnostic”. However, although SQL is quite standard across relational database products, there are product-specific variations and MySQL has a number of non-standard SQL features.

## 6.3 MySQL Preliminaries

For those unfamiliar with the MySQL RDBMS it is advisable to do some preparatory reading and exploratory work before embarking on writing *JDBC* code. When developing with MySQL it is useful to have a MySQL server running on your own computer. That way you can experiment with the schema and query design and then transfer that to your cloud instance. The current stable release of MySQL is version 5.6 and can be downloaded from the [MySQL website](http://www.mysql.com/)<sup>35</sup>. The download includes the *mysql* command line client for interacting with the MySQL database. This client is used in many MySQL tutorials and is quite powerful. It is briefly described below and used in the next section in this chapter.

---

<sup>35</sup><http://www.mysql.com/>

There is a lot of material on the web for learning about MySQL and its dialect of SQL. The most comprehensive is the [MySQL Reference Manual](#).<sup>36</sup> There are also many easily accessible “getting started” type tutorials. In order to get comfortable with MySQL, it is also worth taking time to learn how to use the *mysql* client effectively in order to be able to create databases, switch between databases -a MySQL instance can host multiple databases-, create database objects (tables, views, indexes, etcetera), and run commands to list databases and database objects.

Relational database terminology can be confusing and the same terms can mean different things when applied to different RDBMSs. In the context of MySQL, an **instance** is a set of one or more databases and each database contains a **schema** that defines all its objects (tables, views triggers, relationships, constraints, indexes).

A MySQL-specific oddity is the variety of table types that are available. There are two principal types of table used in MySQL. The first, which is now the default type, is called **InnoDB**. This table type supports database transactions and referential integrity and is the one used in the examples here. The second type called **MyISAM** sacrifices transactions and referential integrity for performance. This table type is not used in any of the examples given here but is commonly used, especially for high performance read-only type applications. A single MySQL database can be composed of multiple table types as suits the user.

## 6.4 Connecting to a Cloud MySQL Database from the *mysql* Client

Once you have set up a MySQL cloud database, you need an application to communicate with the database that is running on the server. That application is called the *client* and there are many client

---

<sup>36</sup><http://dev.mysql.com/doc/refman/5.6/en/>

programs to choose from. Before using *GAS JDBC*, I would first like to briefly review the simplest client of all, the *mysql* command line client. In order to follow along, you will need an accessible MySQL database (it could be running on your local machine or in the cloud). In order to connect to the database you need details of:

- The host where the database server is running
- A user account with permission to connect to the database
- A port number (default is 3306)
- A valid password for the user account

Here is the command I issue from the shell (Mac terminal):

```
1 $ mysql --host=sql5.freemysqlhosting.net --user=sql5622\
2 09 --password
```

In this example, I have not specified a port number so the default 3306 is assumed. On hitting the return key, I am prompted for my password and, once this is accepted, I am connected to the MySQL server and can issue MySQL commands. After a successful login, my terminal displays the following:

```
mmaguire:~ mmaguire$ mysql --host=sql5.freemysqlhosting.net --user=sql562209 --p
assword
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12212762
Server version: 5.5.43-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> █
```

Figure 6-1: *mysql* client when logged in.

Once connected, you can issue commands to the MySQL server. In the log-in command given above no database name was given but you can connect to the database by issuing the command **CONNECT** command passing it the database name as an argument. Figure 6-2 provides a small table of useful *mysql* commands.

Command	Description
STATUS	Display useful information about the server
SHOW DATABASES	Display the names of all available databases on the server
CONNECT <database name>	Connect to a database
SHOW TABLES	Display the names of all tables in the database
DESCRIBE <table name>	Display summary information about the given table
EXIT	Close the mysql client connection

Figure 6-2: Table of useful *mysql* client commands.

From now on, all MySQL commands will be issued from GAS and the code examples will assume that you have set up an accessible MySQL database.

In addition to the *mysql* client, MySQL is well-served by more user-friendly and feature-rich clients. I use the [Sequel Pro](http://www.sequelpro.com/)<sup>37</sup> application (for Macs only) but there is also the very popular multi-platform [MySQL Workbench](https://www.mysql.com/products/workbench/)<sup>38</sup> and the browser application [phpMyAdmin](http://www.phpmyadmin.net/home_page/index.php)<sup>39</sup>. Using any one of these applications should make managing your MySQL database a lot easier than relying on *mysql* alone, especially if you are new to relational databases and MySQL.

## 6.5 An Overview of JDBC

The JDBC API can be used to do very sophisticated database manipulation but the basics are not very difficult, especially for those who already know SQL. By knowing some SQL and how to use a

---

<sup>37</sup><http://www.sequelpro.com/>

<sup>38</sup><https://www.mysql.com/products/workbench/>

<sup>39</sup>[http://www.phpmyadmin.net/home\\_page/index.php](http://www.phpmyadmin.net/home_page/index.php)

small number of JDBC objects and methods, you can accomplish quite a lot. The first, and perhaps most difficult, step is creating a *Connection* object. Once this is created, details on this will follow, you can create a *Statement* object using the *Connection* method *createStatement()*. The *Statement* object does not do anything until you *execute* it using one of three methods:

- *execute()*
- *executeQuery()*
- *executeUpdate()*

Each of these methods are distinguished by the type of SQL statement that they execute. You use the *execute()* method to create database objects, for example, a table or a view. The *executeUpdate()* method does as its name suggests and is used with *UPDATE* and *DELETE* SQL statements and returns the number of rows affected. The method *executeQuery()* is used with *SELECT* SQL statements and returns the database rows as a *ResultSet* object. Think of a *ResultSet* object as an array-of-arrays that you need to process in a nested loop. There is another type of statement called a *PreparedStatement* that is used to pass parameters to the SQL to be executed. I cover some other JDBC objects and methods in the code examples below but these are the main topics that you need to understand in order to use JDBC effectively in GAS. Much of the rest of the material deals with transferring data between database tables and Google Sheets so you will need to know *Sheet* and *Range* objects as discussed in earlier chapters.

## 6.6 Note on Code Examples

All the GAS and SQL code for this chapter is available on [GitHub here](#)<sup>40</sup>, see files “ch06.gs” and “ch06.sql”. The purpose of the code

---

<sup>40</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015)

examples given here is to explain key JDBC concepts so there is a lot of repetition in the example functions. At the end of the chapter I discuss some methods that can be used to make the code more re-usable and more structured. I also plan to add material to later updates of this book to show how certain JavaScript patterns and practices can be used to good effect in larger projects. These approaches are especially important for JDBC where SQL code is embedded within GAS. Embedded SQL makes code maintenance and development quite challenging so any approaches that can be used to deal effectively with this complexity are to be welcomed. The SQL examples also ignore or gloss over very important database concepts like primary and foreign keys, constraints and indexes. Transactions are a standard feature of relational databases and it is important to understand them so they are covered in some detail later in the chapter.

## 6.7 Connecting to the Database

I am assuming that you have set up a Cloud MySQL database, if not, then you will not be able to run the examples given. As with the *mysql* client, in GAS we also need the host name, the user account name, the password and the port number (if it is not the default 3306) in order to establish a connection to the database. Since all examples to follow need a *Connection* object, we can encapsulate *Connection* object creation into a simple GAS function as shown in code example 6.1 below.

### Code Example 6.1



```
1  /**
2  * Return a MySQL connection for given parameters.
3  *
4  * This function will create and return a connection
5  * or throw an exception if it cannot create a
6  * connection using the given parameters.
7  *
8  * @param {String} host
9  * @param {String} user
10 * @param {String} pwd
11 * @param {String} dbName
12 * @param {number} port
13 * @return {JdbcConnection}
14 */
15 function getConnection(host, user, pwd, dbName, port) {
16     var connectionUrl = ('jdbc:mysql://' + host + ':'
17                          + port + '/' + dbName),
18         connection;
19     try {
20         connection = Jdbc.getConnection(connectionUrl, user\
21 , pwd);
22         return connection
23     } catch (e) {
24         throw e;
25     }
26 }
```

This function returns a *Connection* object when called with valid arguments. Note, it expects the database name in addition to the four other arguments. The *getConnection()* method call is enclosed in a *try ... catch* exception handling statement. If anything goes wrong in the *try* block, the error will be raised in the *catch* block. This means that the calling function will have to deal with any errors due to, for example, bad/invalid arguments. In order to get our *Connection* object, we have to call (invoke) the function

`getConnection()` with valid arguments. Code example 6.2 does this.

### Code Example 6.2

```
1  /**
2  * Return a JDBC connection for my
3  * cloud-hosted MySQL
4  *
5  * You will need to modify the variable
6  * values for your own MySQL database.
7  * It calls "getConnection()".
8  *
9  * @return {JdbcConnection}
10 */
11 function getConnectionToMyDB() {
12     var scriptProperties =
13         PropertiesService.getScriptProperties(),
14         pwd =
15             scriptProperties.getProperty('MYSQL_CLOUD_PWD'),
16         user = 'sql562209',
17         dbName = 'sql562209',
18         host = 'sql5.freemysqlhosting.net',
19         port = 3306,
20         connection;
21     connection = getConnection(host, user, pwd, dbName, p\
22 ort);
23     return connection;
24 }
25 /**
26 * Test getConnectionToMyDB()
27 * Prints "JdbcConnection" and closes
28 * the Connection object.
29 *
30 */
31 function test_getConnectionToMyDB() {
```

```
32     var connection = getConnectionToMyDB();
33     Logger.log(connection);
34     connection.close();
35 }
```

The values for all the arguments required by the function *getConnection()* are hard-coded into this function except for the password. The password string is stored as a script property using the key “MYSQL\_CLOUD\_PWD”. I have not discussed script properties but, if you have not used them already, they are a very convenient mechanism for storing data required by GAS scripts. They are very easy to create and retrieve and full documentation is given [here](#)<sup>41</sup>. You will need to change the values for *host*, *database name*, *port* and *password* in the function *getConnectionToMyDB()* to match your own MySQL instance. Once you have done this, you can execute the driver function *test\_getConnectionToMyDB()*. If all goes well, you will see the text “JdbcConnection” in the logging output. Notice that GAS prepends “Jdbc” to all JDBC object names! But, in the interests of brevity and in line with Java JDBC documentation, I have omitted this when discussing these objects.

If you were able to execute the function *test\_getConnectionToMyDB()* using your own MySQL instance, then you have successfully overcome the most difficult obstacle to learning GAS JDBC. The *Connection* object under-pins everything that follows and the function *getConnectionToMyDB()* is used to generate the *Connection* object in all the code examples that follow.

## 6.8 Create, Load, Query, Update and Delete a Database Table

Assuming that you can connect to your MySQL instance from a GAS script and that you have sufficient database permissions, you

---

<sup>41</sup><https://developers.google.com/apps-script/guides/properties>

can now start to do some interesting things. Tables are the fundamental components of relational databases so here is a function that creates a table:

### Code Example 6.3

```
1  /**
2  * Create a JDBC connection and
3  * execute a CREATE TABLE statement.
4  *
5  * return {undefined}
6  */
7  function createTable() {
8      var connection,
9          ddl,
10         stmt;
11     ddl = 'CREATE TABLE people('
12         + 'first_name VARCHAR(50) NOT NULL, \n'
13         + 'surname VARCHAR(50) NOT NULL, \n'
14         + 'gender VARCHAR(6), \n'
15         + 'height_meter FLOAT(3,2), \n'
16         + 'dob DATE)';
17     try {
18         connection = getConnectionToMyDB();
19         stmt = connection.createStatement();
20         stmt.execute(ddl);
21         Logger.log('Table created!');
22     } catch (e) {
23         Logger.log(e);
24         throw e;
25     } finally {
26         connection.close();
27     }
28 }
```

The following examples use the same general structure as this

example so I will explain the code in some depth here. Firstly, the SQL string to execute on the database server is constructed by string concatenation. Unlike the triple quote mechanism in Python or heredocs in other languages, JavaScript does not have a convenient syntax for writing multi-line strings.

All the important action takes place within the *try* block of the *try .. catch .. finally* statement. A *Connection* object is returned by the call to the *getConnectionToMyDB()* function that was defined in code example 6.2.

The *Connection createStatement()* method returns a *Statement* object whose *execute()* method takes the SQL string stored in the variable *ddl* and executes it. The SQL used here is a so-called *Data Definition Language* statement (DDL), hence the name of the variable. DDL statements make some sort of database object, in this example a table is created. If all goes well, the new table will be created and a message to that effect will be printed to the logger. If something goes wrong, the code in the *catch* block will execute so that the error will be printed to the logger and the exception (error) will be thrown. Normally, a database table should have a primary key but this was omitted here to keep the example brief.

Regardless of success or failure, code in the *finally* block is always executed. In this example, the code in the *finally* block is used to close the database connection. This example assumes that a *Connection* object was created in the first place. If this is not so, then an exception will be raised in the *finally* block by calling the *close()* method on a *null* value. Using the *finally* block to release resources is good practice but a real-world application would check the *Connection* object before trying to call the *close()* method.

Before proceeding, it is worth considering what can go wrong in a relatively simple example such as the one discussed above. When our GAS scripts are concerned only with Google Sheets objects such as *Range*, *Sheet* and so on, we have much more control over them but when we are relying on external services such as a MySQL

server, then there is much more scope for things to go wrong. Here are the main ones in my experience:

- Incorrect connection details (host name, port, or password)
- The server is not running (the server is “down”)
- The account being used has insufficient privileges
- SQL syntax is incorrect
- Trying to create an object that already exists or trying to drop an object that does not exist
- The SQL to execute violates a database constraint

Now that the table is created, we can add some data to it by executing an SQL *INSERT* statement as show below:

#### Code Example 6.4

```
1 // Code example 6.4
2 /**
3  * Execute a single SQL INSERT statement to
4  * add the table created by code example 6.3.
5  *
6  * @return {undefined}
7  */
8 function populateTable() {
9     var connection,
10         stmt,
11         sql,
12         rowInsertedCount;
13     sql = "INSERT INTO people(first_name, surname, gender\
14 , height_meter, dob) VALUES\n"
15         + "('John', 'Browne', 'Male', 1.81, '1980-0\
16 5-03'),\n"
17         + "('Rosa', 'Hernandez', 'Female', 1.70, '1\
18 981-04-30'),\n"
```

```

19             + "('Mary', 'Carr', 'Male', 1.72, '1982-08-\
20 01'),\n"
21             + "('Lee', 'Chang', 'Male', 1.88, '1973-07-\
22 15')";
23     try {
24         connection = getConnectionToMyDB();
25         stmt = connection.createStatement();
26         rowInsertedCount = stmt.executeUpdate(sql);
27         Logger.log(rowInsertedCount + ' values inserted!');
28     } catch (e) {
29         Logger.log(e);
30         throw e;
31     } finally {
32         connection.close();
33     }
34 }

```

The code structure in this example is very similar to the one in the previous example. The SQL statement is created by string concatenation and the database interaction is performed in a *try* block. This SQL statement *creates* rows in a target database table using the SQL *INSERT* statement. This is the “C” in the “CRUD” acronym (Create, Read, Update, Delete). The code creates a *Statement* object as in the previous example. The *INSERT* statement is executed using the *executeUpdate()* method rather than the plain *execute()* that was used earlier and the message “4 values inserted!” is printed to the logger. Note, the *execute()* method also works but *executeUpdate()* returns the number of rows inserted. This return value is especially useful for updates and deletes. One this function is executed, the four inserted rows will be visible in your preferred MySQL client.

To see the inserted rows, we execute a *SELECT* statement (the R for Read in CRUD):

### Code Example 6.5

```
1  /**
2  * Execute a SELECT statement to
3  * return all table rows
4  *
5  * Print rows to log viewer.
6  * @return {undefined}
7  */
8  function selectFromTable() {
9      var connection,
10         stmt,
11         sql,
12         rs,
13         rsmd,
14         i,
15         colCount;
16     sql = 'SELECT * FROM people';
17     try {
18         connection = getConnectionToMyDB();
19         stmt = connection.createStatement();
20         rs = stmt.executeQuery(sql);
21         rsmd = rs.getMetaData();
22         colCount = rsmd.getColumnCount();
23         while(rs.next()) {
24             for(i = 1; i <= colCount; i += 1) {
25                 Logger.log(rs.getString(i) + ', ');
26             }
27             Logger.log('\n');
28         }
29     } catch (e) {
30         Logger.log(e);
31         throw e;
32     } finally {
33         connection.close();
34     }
35 }
```



When this function is executed, the results are printed to the logger. The code should now look familiar but this example introduces two new objects and the *Statement* execute method used is *executeQuery()* (trying to use either *execute()* or *executeUpdate()* in this context will generate an error). The method *executeQuery()* returns a *ResultSet* object that represents the rows returned by the SQL statement. The “SELECT \*” construct returns all columns. In order to process the *ResultSet* in the nested loop, we need to know how many columns were returned. This information is available from the *MetaData* object returned by the *ResultSet* *getMetaData* method. The outer *while* loop tests the *ResultSet* *next()* method and with each call a **cursor** moves forward by one row. The cursor acts like a pointer into the record set, when there are no more rows, the call to *next()* returns *false* and the *while* loop exits. The inner *for* loop processes each column in the current row and the *ResultSet* *getString()* method is used to extract the column value for that row. There are a number of other *RecordSet* get methods in JDBC but, since JavaScript (unlike Java,) is loosely typed, *getString()* works for dates and numbers as well as for strings.

Now we have a table with some rows that we can view. Sometimes we need to change rows in the table, the “U” for “Update” in “CRUD”. Our original data gave the gender for the person named “Mary Carr” as “Male”, the following code updates the gender to “Female”.

### Code Example 6.6

```
1  function updateTable() {
2      var connection,
3          stmt,
4          sql,
5          updateRowCount;
6      sql = "UPDATE people SET gender = 'Female'\n"
7           + "WHERE first_name = 'Mary' AND surname = 'Carr\
8  ";
9      try {
10         connection = getConnectionToMyDB();
11         stmt = connection.createStatement();
12         updateRowCount = stmt.executeUpdate(sql);
13         Logger.log(updateRowCount + ' rows updated!');
14     } catch (e) {
15         Logger.log(e);
16         throw e;
17     } finally {
18         connection.close();
19     }
20 }
```

As with the *INSERT* example given earlier, we use the *executeUpdate()* method so that we can determine how many rows were updated.

The last of the “CRUD” operations is “D” for “Delete”. In the following example, we delete the row where the name is “Lee Chang”:

### Code Example 6.7

```
1  /**
2  * Delete a row from the database
3  *
4  * Print the deleted row count to
5  * the log.
6  *
7  * @return {undefined}
8  */
9  function deleteTableRow() {
10     var connection,
11         stmt,
12         sql,
13         deleteRowCount;
14     sql = "DELETE FROM people\n"
15         + "WHERE first_name = 'Lee' AND surname = 'Chang\
16     '";
17     try {
18         connection = getConnectionToMyDB();
19         stmt = connection.createStatement();
20         deleteRowCount = stmt.executeUpdate(sql);
21         Logger.log('Deleted row count = ' + deleteRowCount);
22     } catch (e) {
23         Logger.log(e);
24         throw e;
25     } finally {
26         connection.close();
27     }
28 }
```

The explanation given for example 6.6 applies to this example also. The only difference is that this time the SQL statement to execute is a *DELETE* in place of *UPDATE*.

Finally, now that we have covered the CRUD operations, we can clean up the database by dropping the table we have been experi-

menting with. Here is the code to do just that:

### Code Example 6.8

```
1  /**
2  * Drop the table from the database
3  *
4  * @return {undefined}
5  */
6  function dropTable() {
7      var connection,
8          stmt,
9          ddl;
10     ddl = 'DROP TABLE IF EXISTS people';
11     try {
12         connection = getConnectionToMyDB();
13         stmt = connection.createStatement();
14         stmt.execute(ddl);
15         Logger.log('Table "people" dropped!');
16     } catch (e) {
17         Logger.log(e);
18         throw e;
19     } finally {
20         connection.close();
21     }
22 }
```

We created the table using the *CREATE TABLE* DDL statement and now we drop it, and all the data it contains, using the *DROP TABLE* statement. MySQL includes the useful *IF EXISTS* clause so we can issue *DROP* statements without first having to determine if the object in question exists.

The examples above cover the very basics of JDBC but in all examples the SQL strings were represented as hard-coded variables. What about situations when we want to *SELECT*, *UPDATE* or

*DELETE* just a sub-set of rows in a table? Each of these three SQL statements supports a *WHERE* clause that we can use to refine the operation so that it only applies to a sub-set of rows. We could generate dynamic SQL statements by, for example, using function arguments and string concatenation but there is a better and safer approach. We can use **parameterized queries** and these are implemented in JDBC using a type of *Statement* object called a *PreparedStatement*.

## 6.9 Prepared Statements

Prepared statement objects are very similar to the *Statement* objects that were described earlier. They are created from *Connection* objects and they implement the same methods for executing SQL statements. In fact, they could have been used as a drop-in replacement for *Statement* objects in all the examples given so far. If the two statement types are so similar, why bother with the *prepared* version. There are two reasons to use the *prepared* version:

- The *prepared* version can be pre-compiled
- The *prepared* version is more secure

Pre-compiling the SQL statement can allow the RDBMS to perform what Oracle refers to as a “soft parse”. For example, given this type of SQL statement:

```
1 SELECT ename FROM emp WHERE empno = ?
```

The “?” represents a parameter that is to be provided when the SQL is executed and it might represent a value passed in by the user from some sort of GUI or a spreadsheet cell value. The important point is that the value is unknown before the code executes so it cannot be hard-coded. The naive way to pass in the value would be to do

some sort of string manipulation and then execute the completed SQL using one of the *Statement* methods discussed earlier. While this will work it is both inefficient and error prone. If the value is not numeric, then you need to handle the enclosing single quotes. It is inefficient because the RDBMS will re-parse the entire SQL statement for every new “empno” value. However, when a prepared statement is used, the RDBMS can parse it just once. When running thousands of these types of queries, the performance benefits can be considerable.

The prepared statement is more secure because it helps guard against so-called [SQL injection attacks](#)<sup>42</sup>. In this type of attack, a malicious user could substitute destructive SQL in place of the “?”. If the convenience and performance benefits of using prepared statements for the type of SQL statement given above do not convince you, then the threat of an SQL injection attack should.

When prepared statements are used, the value used to replace the “place-holder” is referred to as a **bind variable**. Bind variables are widely used and the same principle is involved in VBA ADO using the *CreateParameter()* method.

Having discussed them, it is now time to show some code that uses prepared statements. To run these examples, you will need to execute the SQL code given in [this GitHub location](#)<sup>43</sup>. When run, it will create two tables called “emp” and “dept”. The tables and their data are taken from a very old Oracle training schema called “scott”.

### Code Example 6.9

---

<sup>42</sup>[https://en.wikipedia.org/wiki/SQL\\_injection](https://en.wikipedia.org/wiki/SQL_injection)

<sup>43</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015/blob/master/ch06.](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch06.sql)

```
1  /**
2  * Demonstrate the use of a prepared statement.
3  *
4  * Prints the name of an employee for a
5  * given parameter.
6  * @return {undefined}
7  */
8  function runPrepStmt() {
9      var connection,
10         pStmt,
11         sql,
12         rs,
13         empName;
14     sql = 'SELECT ename FROM emp WHERE empno = ?';
15     try {
16         connection = getConnectionToMyDB();
17         pStmt = connection.prepareStatement(sql);
18         pStmt.setInt(1, 7839);
19         rs = pStmt.executeQuery();
20         rs.next();
21         empName = rs.getString(1);
22         Logger.log(empName);
23     } catch (e) {
24         Logger.log(e);
25         throw e;
26     } finally {
27         connection.close();
28     }
29 }
```

This code example uses a *PreparedStatement* object with a single bind variable represented by the “?” for the employee number. The *PreparedStatement* object is returned by the method *prepareStatement()*. Unlike when the plain *Statement* object was used, the SQL string argument is passed when the *PreparedStatement* object is

created and not when the *executeQuery()* method is called. The “?” is the place-holder for the bind parameter and the bind parameter value is set by one of the *set* methods (*setInt()* in this example). The bind parameter values are set in order of place-holder appearance in the SQL statement starting at one. As with the plain *Statement* object, the *executeQuery()* method returns a *ResultSet* object. In this example we know that the query can only return one row because the bind parameter corresponds to the primary key for the target table and we are only returning one column. Therefore, we do not need a nested loop to process the *ResultSet*, we just need to move the cursor forward (*rs.next()*) followed by a single *getString()* call.

Prepared statements can also be used very effectively with SQL *UPDATE* and *DELETE* statements. There are many *set* methods for assigning values to the bind parameters with the most useful being *setString()*, *setDate()* and *setFloat()*, see [here](#)<sup>44</sup> for the full list. MySQL column types are strict so you need to use the correct *set* method for the target column in the database.

## 6.10 Transactions

A very important feature of relational databases is transactions. Transactions are an “all-or-nothing” proposition; They comprise a group of SQL statements that either proceed to completion without error or are rejected entirely such that they have no effect on the state of the database. Transactions are the means by which RDBMs implement the **ACID**<sup>45</sup> set of properties (Atomic, Consistent, Isolated, Durable). The [Wikipedia entry](#)<sup>46</sup> gives a good overview.

When discussing transactions two important concepts are committing and rolling back. When a database change is committed, it is made permanent and will be available to other users. Up

---

<sup>44</sup><http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html>

<sup>45</sup><https://en.wikipedia.org/wiki/ACID>

<sup>46</sup>[https://en.wikipedia.org/wiki/Database\\_transaction](https://en.wikipedia.org/wiki/Database_transaction)



to this point all the CRUD examples given have been committed automatically to the database. This is fine when the database operations are simple but in more complex applications transactions become necessary. The default setting in Google JDBC is to commit automatically but this behaviour can be changed. The following code example is rather contrived but it shows how to turn off the auto-commit, do an SQL update and then undo (roll back) the change.

### Code Example 6.10

```
1  /**
2  * Show effect if switching off auto-commit.
3  *
4  * Give employee number 7900 ("JAMES") a
5  * 100-fold pay increase!
6  * But the change is "rolled back".
7  * Try changing "connection.rollback()" to
8  * "connection.commit()" and see the effect.
9  */
10 function demoTransactions() {
11     var connection,
12         stmt,
13         sql = 'UPDATE emp SET sal = sal * 100 WHERE empno\
14 = 7900';
15     connection = getConnectionToMyDB();
16     connection.setAutoCommit(false);
17     stmt = connection.createStatement();
18     stmt.executeUpdate(sql);
19     connection.rollback();
20     connection.close();
21 }
```

The *Connection* method *setAutoCommit()* can be used to alter auto-commit behaviour. In this example auto-commit is set to *false* and

an SQL *UPDATE* statement is issued to increase the salary for empno 7900 100-fold. Although the update statement is executed, its effect is nullified by the *rollback()* method call. If a *commit()* call had been issued instead, then the change would have been made permanent. For applications that are performing multiple, inter-dependent database changes, it is advisable to turn off auto-commit and use transactions where explicit commits or rollbacks are issued as required.

## 6.11 Database Metadata

Metadata, that is data about data, is stored in the database data dictionary. MySQL uses a series INFORMATION\_SCHEMA tables to store metadata. These tables can be queried directly using SQL to extract useful information about tables, their definitions, database privileges, column names, column data types and other information. When working with a complex database with many tables, a good entity-relationship diagram tool is very useful for showing the overall schema with the tables and the relationships between them. MySQL Workbench is excellent in this respect. To get more detail on tables, their columns, column data types and constraints, you can query the INFORMATION\_SCHEMA tables. The following SQL query extracts some useful information about all the tables and the columns of those table

### Code Example 6.11

```

1  SELECT
2    tab.table_name,
3    tab.engine,
4    tab.table_rows,
5    col.column_name,
6    col.column_type
7  FROM
8    INFORMATION_SCHEMA.TABLES AS tab
9    JOIN INFORMATION_SCHEMA.COLUMNS AS col
10     ON tab.table_name = col.table_name
11 WHERE
12     tab.table_type = 'BASE TABLE';

```

you can also extract database metadata using JDBC object methods. Metadata has already been extracted in previous example (Code Example 6.5) to determine the result set column count when the method `getMetaData()` was invoked on the `ResultSet` object. The `getMetaData()` is also implemented by the `Connection` object. The following function uses a `Connection` object to obtain and print some useful database information.

### Code Example 6.12

```

1  /**
2   *Print some information about
3   * the target database to the logger.
4   *
5   * @return {undefined}
6   */
7  function printConnectionMetaData() {
8     var connection = getConnectionToMyDB(),
9         metadata = connection.getMetaData();
10     Logger.log('Major Version: ' + metadata.getDatabaseMa\
11     jorVersion());
12     Logger.log('Minor Version: ' + metadata.getDatabaseMi\

```

```
13 norVersion());
14   Logger.log('Product Name: ' + metadata.getDatabasePr\
15   oductName());
16   Logger.log('Product Version: ' + metadata.getDatabase\
17   ProductVersion());
18   Logger.log('Supports transactions: ' + metadata.supp\
19   rtsTransactions());
20 }
```

A *MetaData* object is returned from the *Connection* object (created by *getConnectionToMyDB()* defined in code example 6.2) by calling the *getMetaData()* method. We can then interrogate the *MetaData* object using its methods. If you execute this function and then check the logger output, you should see some useful information.

## 6.12 A Practical GAS Example

We have now covered the main JDBC objects and methods but there is a lot of repetition of code. Apart from creation of the *Connection* object, no other database operations have been written to be re-usable. Of course you can copy-and-paste and then edit code snippets to suit your purpose but this only really works for trivial examples. Most database-backed application involve a complex inter-play between application code and the database. In order to make the code manageable, it really helps to structure it so that repetition is minimised.

The function is the basic unit of code re-use in JavaScript so we can begin to structure our database application code using functions. This is the approach we will use here to implement a series of GAS functions for executing a SQL statement, processing the results and writing them to a sheet. That is a fairly standard type of data flow in a database-backed application. Here are the individual steps:

- Create a database connection

- Execute a *SELECT* statement with a bind parameter and return a *ResultSet*
- Get the column names for the *ResultSet* as an array
- Get the data rows of the *ResultSet* as an array-of-arrays
- Create a new sheet
- Write the column names and data row values to the new sheet

I am going to use the *scott* database with its two tables called *dept* and *emp*, see the entity-relationship diagram in figure 6-3 below.

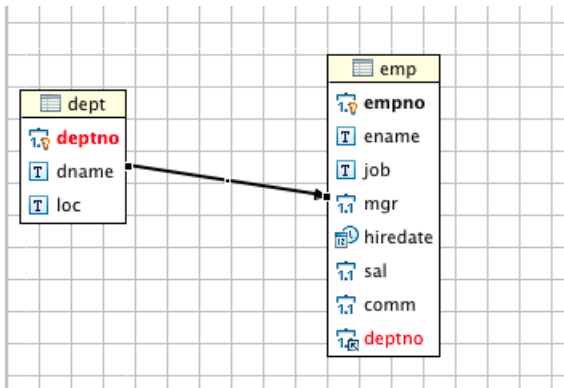


Figure 6-3: Entity-Relationship Diagram for *scott* schema.

I will re-use the code examples 6.1 and 6.2 to create the database *Connection* object. The first task is to create a *ResultSet* object as show in this function:

### Code Example 6.13

```

1  /**
2  * Return a ResultSet
3  *
4  * @param {JdbcConnection} conn
5  * @param {number} deptno
6  * @return {JdbcResultSet}
7  */
8  function getResSetForDept(conn, deptno) {
9      var sql = ('SELECT\n' +
10                ' e.ename, \n' +
11                ' e.hiredate, \n' +
12                ' e.job, \n' +
13                ' e.sal, \n' +
14                ' d.dname\n' +
15                'FROM\n' +
16                ' emp e\n' +
17                ' JOIN dept d ON e.deptno = d.deptno\n' +
18                'WHERE\n' +
19                ' d.deptno = ?'),
20      pstmt = conn.prepareStatement(sql),
21      rs;
22      pstmt.setInt(1, deptno);
23      recset = pstmt.executeQuery();
24      return recset;
25 }

```

This function uses the *Connection* object argument to create and return a *ResultSet* using the embedded SQL and the bind parameter *deptno*.

The *ResultSet* object returned by the function above encapsulates both the query output rows and metadata. The next function takes the *ResultSet* object as an argument and queries its metadata to extract the column names.

**\*\* Code Example 6.14 \*\***

```
1  /**
2  * Return the column names for
3  * given ResultSet
4  * @param {JdbcResultSet} rs
5  * @return {String[]}
6  */
7  function getColNames(rs) {
8      var md = rs.getMetaData(),
9          colCount = md.getColumnCount(),
10         colNames = [],
11         colName,
12         i;
13     for(i = 1; i <= colCount; i +=1) {
14         colName = md.getColumnName(i);
15         colNames.push(colName);
16     }
17     return colNames;
18 }
```

This function extracts the *MetaData* object from the *ResultSet* argument and uses a *for* loop to build an array of column names that it then returns.

Having extracted the column names from the *ResultSet*, we now want the data row values. The next functions returns them as an array-of-arrays.

#### Code Example 6.15

```
1  /**
2  * Return the data rows as an array-of-arrays
3  * for the given ResultSet
4  * @param {JdbcResultSet} rs
5  * @return {String[][]}
6  */
7  function getDataRows(rs) {
8      var md = rs.getMetaData(),
9          colCount = md.getColumnCount(),
10         row = [],
11         value,
12         dataRows = [],
13         colName,
14         i;
15     while(rs.next()) {
16         row = [];
17         for(i = 1; i <= colCount; i +=1) {
18             value = rs.getString(i);
19             row.push(value);
20         }
21         dataRows.push(row);
22     }
23     return dataRows;
24 }
```

This function loops over the *ResultSet* argument with the outer *while* loop moving the cursor forward through the rows and the inner *for* loop extracting the column values. Each row is represented as an array and the rows for the entire *ResultSet* are returned as an array-of-arrays.

We now have all the data we need from the database and the remaining functions deal with the Google Sheet part of the application. We have two arrays, one with column names that we will use as Sheet column headers and the other with the data values. All



the GAS objects and methods below relating to *Sheet* and *Range* objects were discussed in earlier chapters so the coverage here will be brief.

The following is a general utility function that takes a sheet name argument and returns a boolean value to indicate if that sheet name exists in the active spreadsheet (assumed to be the one from which the GAS code is executed).

**\*\* Code Example 6.16 \*\***

```
1  /**
2  * Check if given sheet name
3  * exists in the active spreadsheet.
4  * @param {String} sheetName
5  * @return {boolean}
6  */
7  function sheetExists(sheetName) {
8      var ss = SpreadsheetApp.getActiveSpreadsheet(),
9          sheets = ss.getSheets(),
10         i;
11     for(i = 0; i < sheets.length; i += 1) {
12         if( sheets[i].getSheetName() == sheetName) {
13             return true;
14         }
15     }
16     return false;
17 }
```

Next up is another general utility GAS function that deletes the sheet with the given name, replaces it with a new blank sheet with the same name and returns the *Sheet* object.

**\*\* Code Example 6.17\*\***

```

1  /**
2  * Delete sheet with given name
3  * and replace it with new one
4  * with same name.
5  * @param {String} sheetName
6  * @return {Sheet}
7  */
8  function replaceOldSheet(sheetName) {
9      var ss = SpreadsheetApp.getActiveSpreadsheet(),
10         sh = ss.getSheetByName(sheetName);
11         ss.deleteSheet(sh);
12         sh = ss.insertSheet(sheetName);
13         return sh
14     }

```

The last general utility function inserts a new sheet into the active spreadsheet with the given sheet name.

**\*\* Code Example 6.18\*\***

```

1  /**
2  * Create a sheet with the given name
3  * in the active spreadsheet
4  * @param {String} sheetName
5  * @return {Sheet}
6  */
7  function addNewSheet(sheetName) {
8      var ss = SpreadsheetApp.getActiveSpreadsheet(),
9         sh = ss.insertSheet(sheetName);
10         return sh;
11     }

```

Now we have all the functions we need to implement the functionality that was specified earlier. We just need a **driver function** to make calls to these functions in the correct sequence and we should

see the output as a new sheet filled with data from the database. This function is called *main* and is defined below.

**\*\* Code Example 6.19 \*\***

```
1  /**
2  * Run application to extract a set of
3  * rows from the database and write them
4  * to a new sheet.
5  * @return {undefined}
6  */
7  function main() {
8      var conn = getConnectionToMyDB(),
9          deptno = 20,
10         rs = getResSetForDept(conn, deptno),
11         colNames = getColNames(rs),
12         dataRows = getDataRows(rs),
13         sheetName = 'RESEARCH',
14         sh,
15         dataRowCount = dataRows.length,
16         dataColCount = colNames.length,
17         outputHeaderRng,
18         outputDataRng;
19     if(sheetExists(sheetName)) {
20         sh = replaceOldSheet(sheetName)
21     } else {
22         sh = addNewSheet(sheetName);
23     }
24     outputHeaderRng = sh.getRange(1,1,1,dataColCount);
25     outputHeaderRng.setFontWeight('bold');
26     outputDataRng = sh.getRange(2,
27                             1,
28                             dataRowCount,
29                             dataColCount);
30     outputHeaderRng.setValues([colNames]);
```

```
31  outputDataRng.setValues(dataRows);
32  rs.close();
33  conn.close();
34 }
```

This function is quite straight-forward and omits exception handling to keep it as brief as possible. It creates and populates two arrays, one for the column names and the second for the data values. It creates a new sheet and defines two *Range* objects for this sheet (*sh.getRange()*). This four argument version of the *Sheet getRange()* method was explained in the previous chapter. The *Range setValues()* method takes an array-of-arrays as its argument and writes the data to the target range. Note the trick to write the one-dimensional *colNames* array to the range. It is enclosed in a “[ ]” to make it an array with one element that is another array. If all goes well, you should see output as shown in figure 6-3 below.

A	B	C	D	E
ename	hiredate	job	sal	dname
SMITH	12/17/1980	CLERK	800	RESEARCH
JONES	4/2/1981	MANAGER	2975	RESEARCH
SCOTT	12/9/1982	ANALYST	3000	RESEARCH
ADAMS	1/12/1983	CLERK	1100	RESEARCH
FORD	12/3/1981	ANALYST	3000	RESEARCH

Figure 6-4: Database output in a new sheet.

The code examples given here should provide ideas on how to break your application into small re-usable functions that separate the database-specific code from the code that manipulates *Sheet* and *Range* objects. While these examples work, the application could be greatly improved by using JavaScript’s object oriented capabilities. I will return to this topic in a chapter that will be added to this book in the future. In the meantime, I have written some blog entries that describe JavaScript patterns that could be used: The [Construc-](#)

tor/Prototype pattern<sup>47</sup> and the Revealing Module pattern<sup>48</sup>. The Revealing Module example can be used to convert Google Sheets data into executable SQL for either SQLite, MySQL or PostgreSQL. It creates both the DDL (*CREATE TABLE*) and the *INSERT INTO* for that table using the column headers in the spreadsheet for the column names in the table. If you spend a lot of time and effort transferring Google Sheets data into one of these RDBMSs then you might want to check it out.

To use JDBC GAS requires SQL. There are two potential difficulties with this. Firstly, it requires the programmer to know SQL. Secondly, embedding SQL statements in GAS is a bit of a pain and JavaScript's lack of support for convenient multi-line strings does not help. This problem becomes more acute when the SQL is big and complex. There are a few ways to ameliorate this problem. One is to use views (stored queries) that are stored on the database. That way you can create a complex view, test it, and document so that JDBC GAS can use it as the target for *SELECT* statements. A second alternative is to create *stored procedures* that can be called by JDBC GAS. In this way you can delegate all the complex SQL stuff to the stored procedure and just call it with the required arguments. This requires that you know the stored procedure language for your particular RDBMS. Different RDBMSs use different languages with different capabilities, unfortunately. The cloud MySQL service that I use does not grant me privileges to create either views or stored procedures so I have not given examples here. You will need to check if using these approaches is allowed by the provider you have chosen.

All the GAS code examples in this chapter use code executed from the script editor. In real-life database-backed applications a graphical user interface (GUI) is usually provided and that is the subject of the next chapter.

---

<sup>47</sup><http://www.javascript-spreadsheet-programming.com/2013/01/object-oriented-javascript-part-2.html>

<sup>48</sup><http://www.javascript-spreadsheet-programming.com/2015/04/automated-generation-of-sql-from-google.html>

## 6.13 Summary

This chapter has covered a lot of material on using JDBC in GAS. It requires that you have access to a Cloud-hosted MySQL database and that you know the basics of SQL. Having connected to the database, you can (subject to permissions) execute the full range of SQL statements to create database objects, add data to tables, update this data and extract it from the database into Google Sheets. As your data volume and user base both grow, the advantages of delegating data storage to a relational database become clear and the time and effort required to learn SQL and GAS JDBC become very worthwhile.

# Chapter 7: User Interfaces - Menus and Forms

## 7.1 Introduction

All the code examples in the book up to this point are executed from the GAS script editor. Executing code in this way makes sense when learning GAS but an application intended for end-users requires some sort of Graphical User Interface (GUI) if it is to be successful. Ideally, the GUI should hide the GAS entirely and allow users to use the application without concerning themselves with how the application is implemented. This chapter covers the basics of creating simple menus and forms. Forms will be implemented using *HtmlService*. The earlier version of this book also covered the older *UiApp* technology for building GUIs but this approach has now been deprecated by Google and is not discussed any further [here](#)<sup>49</sup>.

*HtmlService* uses HTML5, Cascading Style Sheets (CSS) and client-side JavaScript (JavaScript executed by the browser) to build feature-rich web applications. Anyone with web development experience will feel very comfortable using *HtmlService*. Because the topic is so wide and involves HTML, CSS and JavaScript, this chapter will cover only the basics of creating forms that can be used to write data to spreadsheets and display spreadsheet data.

All the code for the examples given in this chapter is available on the book's [GitHub](#)<sup>50</sup> repository. There are two files relating to this chapter: `ch07.gs` and `ch07.html`.

---

<sup>49</sup><https://developers.google.com/apps-script/reference/ui/ui-app>

<sup>50</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015)

## 7.2 Adding A Menu

The simplest way to execute GAS code is to add a menu to the Google Sheets menu bar. Menu item selection can then execute a GAS function. Placing a new menu on the menu bar is done using GAS. How then, you may ask, is the GAS that builds and populates the menu executed? It is executed using what Google refer to as a **trigger**. A trigger is a GAS function that executes in response to an event. Google Sheets pre-defines a small set of [trigger functions](#)<sup>51</sup> and the one we will use extensively in this chapter is called `onOpen()`. Each Google Sheet has its own definition of this function trigger and it executes every time a Google Sheet is opened. We can use it as a “hook” to define what should happen when the Google Sheet is opened. One very important use of the `onOpen()` trigger function is provide a mechanism to display menus and forms. Before discussing forms, let’s create a simple and useless menu and add two items to it.

### Code Example 7.1

```
1 // Code Example 7.1
2 /*
3  Function "onOpen()" is executed when a Sheet
4  is opened. The "onOpen()" function given here
5  adds a menu to the menu bar and adds two items
6  to the new menu. Each of these items execute a
7  function when selected. The functions simply display a
8  message box dialog.
9  */
10 function onOpen() {
11     var ss = SpreadsheetApp.getActiveSpreadsheet(),
12         menuItems = [{name: "Item1",
13                     functionName: "function1"}],
```

---

<sup>51</sup><https://developers.google.com/apps-script/guides/triggers/>



```

14         {name: "Item2",
15           functionName: "function2"}];
16     ss.addMenu("My Menu", menuItems);
17 }
18
19 function function1() {
20     Browser.msgBox('You selected menu item 1');
21 }
22
23 function function2() {
24     Browser.msgBox('You selected menu item 2');
25 }

```

Create a new Google Sheet and add the code example 7.1 to the GAS script editor and then save, close and re-open the Google Sheet. When opened you should see a new menu entitled “My Menu” and when selected, you will see something like figure 7-1 below.

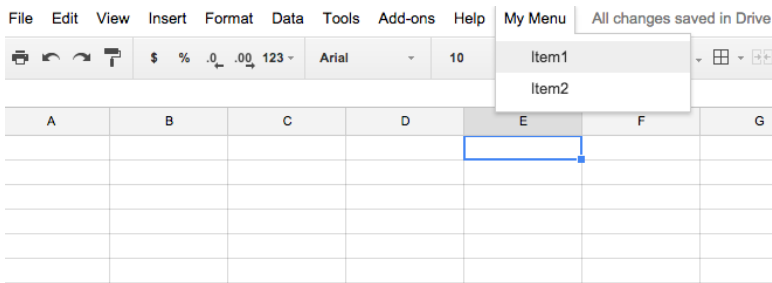


Figure 7-1: A custom-built menu added to the menu bar

Selecting one of the items displays a message box. So now that we have a graphical method for executing GAS code we can build something more useful. When we come to discussing GUI forms the same approach will be used to create a menu for displaying when the Google Sheets spreadsheet is opened and a description of the code used here will help prepare for that. The menu is created, populated with menu items and added to the Sheets menu bar by the

*onOpen()* function. There are a number of different mechanisms for building the menu but I think the one used here is the easiest where a menu title and an array of objects are passed to the *Spreadsheet* object *addMenu()* method. Each object in the array has two items with keys “name” and “functionName” for, respectively, the text to be displayed for the menu item and the name of the function to be executed when the menu item is selected. Note that the function name has to be given in quotes (double or single) and without parentheses. That is all I am going to discuss about menus here but, if you would like to know more, read the [Google documentation](#)<sup>52</sup>. Menus can be used to execute complex code and, in conjunction with input boxes and message boxes, they may suffice for simple applications. More complex applications usually require a form of some sort and forms are discussed in the next section.

## 7.3 Building Forms With *HtmlService*

One of the really nice features of Excel VBA is the ease with which you can build quite complex forms using the form designer and its drag-and-drop functionality. Google’s now deprecated *UiApp* had a GUI Builder that provided a similar experience. *HtmlService* is now the approved way to build forms for Google Sheets. It doesn’t offer any nice-and-easy drag-and-drop form designer but instead it offers something altogether more powerful; it offers the full power of HTML5, CSS and client-side JavaScript so that we can build web applications to interact with and control our Google spreadsheet. The three technologies have distinct roles in web development:

1. HTML implements the application structure and provides the elements with which the user interacts (text boxes, buttons and the like).

---

<sup>52</sup><https://developers.google.com/apps-script/guides/menus?hl=en>

2. CSS is concerned with the presentation and is used to set positions of elements on the form as well as their colours, fonts and so on.
3. JavaScript provides the behaviour of the application, for example, it responds to user interaction such as button clicks.

In order build feature-rich web application for Google Sheets, you have to learn at least the basics of these three technologies. Anyone who has worked through this book up to this point should be proficient in GAS and therefore will know quite a bit of JavaScript already. The basics of HTML and CSS are not difficult but both subject areas are very large so they will only be covered briefly here. My hope is that the coverage will be sufficient to provide motivated readers with enough expertise and knowledge to get started. All the examples given have been **tested in Google Chrome only**. They should also display as intended in recent versions of Safari, Firefox and Internet Explorer. Some of the HTML5 elements that I demonstrate may not work in some browsers but I have only included those elements that I know are implemented by Google Chrome.

### 7.3.1 A Simple Data Entry Form

One of the most common uses of forms in spreadsheet application is for data entry. We will build a very simple form that takes two text values and writes them to cells in the spreadsheet hosting the form. We will also add a menu to launch the form that uses the *onOpen()* trigger function to create the menu and add it to the menu bar as discussed in the previous section. Since this section is our introduction to HtmlService, I will discuss the code in some detail. For this and subsequent examples there are two source code files:

1. Standard GAS source file.
2. HTML source file.

The HTML file is created in the GAS script editor by the menu action *File->new->Html file*. Here is the code for the HTML and GAS files:

### Code Example 7.2

```
1  // Code Example 7.2
2  /*
3   Create a simple data entry form that collects
4   two text values and writes them to the active sheet.
5   See also "index.html" in code example ch07.html
6   */
7
8  /*
9   Trigger function to build a menu and add it
10  to the menu bar when the spreadsheet is opened.
11  */
12  function onOpen() {
13    var ss = SpreadsheetApp.getActiveSpreadsheet(),
14        menuItems = [{name: "Show Form",
15                      functionName: "showForm"}];
16    ss.addMenu("Data Entry", menuItems);
17  }
18  /**
19   * Read in the Html from file "index.html"
20   * and create and display the form.
21   */
22  function showForm() {
23    var ss = SpreadsheetApp.getActiveSpreadsheet(),
24        html = HtmlService
25            .createHtmlOutputFromFile('index')
26            .setSandboxMode(HtmlService
27                .SandboxMode.IFRAME);
28    ss.show(html);
29  }
```

```
30
31  /*
32   Take a form as an argument and use the input element n\
33   ame
34   attribute to reference the text values entered by the \
35   user
36   and write these values to the active spreadsheet.
37  */
38  function getValuesFromForm(form){
39    var firstName = form.firstName,
40        lastName = form.lastName,
41        sheet = SpreadsheetApp.getActiveSpreadsheet()
42                .getActiveSheet();
43    sheet.appendRow([firstName, lastName]);
44  }
```

Here is the HTML to go with the above GAS file (named “index.html”):

```
1  <!--
2   Code Example 7.2
3   index.html
4   -->
5  <div>
6    <b>Add Row To Spreadsheet</b><br />
7    <form>
8      First name: <br />
9      <input id="firstname" name="firstName"
10         type="text" />
11     <br />
12     Last name: <br />
13     <input id="lastname" name="lastName"
14        type="text" />
15     <br />
```

```
16     <input onclick="formSubmit()" type="button"
17         value="Add Row" />
18     <input onclick="google.script.host.close()"
19         type="button" value="Exit" />
20 </form>
21 <script type="text/javascript">
22     function formSubmit() {
23         google.script.run
24             .getValuesFromForm(document.forms[0]);
25     }
26 </script>
27 </div>
```

After pasting in the code, close the spreadsheet and re-open it. The new menu entitled “Data Entry” should appear on the right-hand side of the menu bar. On the menu’s “Show Form” item, the following form should appear:

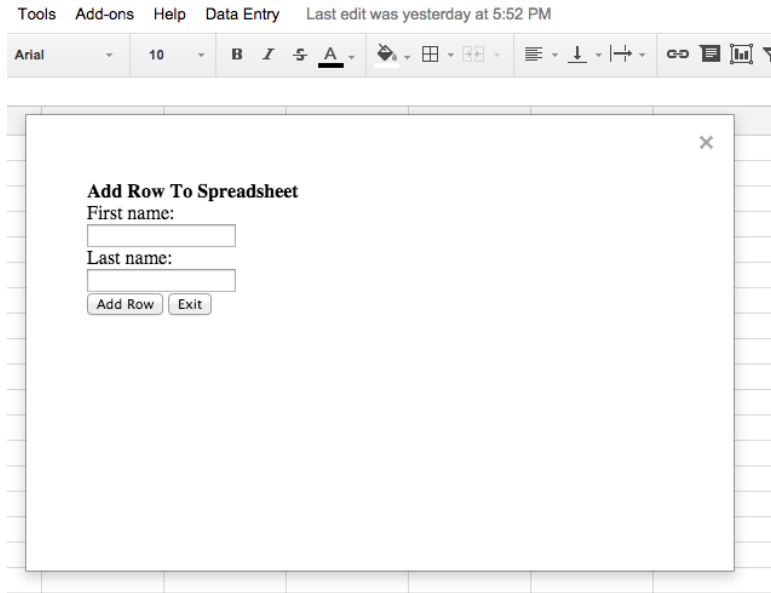


Figure 7-2: Basic data entry form.

Try out the form by entering some values in the text boxes and click the “Add Row” button to see the values transferred to the active sheet! You can close the form by clicking the “Exit” button.

The form itself is very basic, it uses the browser defaults for layout, style and size and uses only the HTML text input element for data entry. I will talk more about layout and styling later. I will explain this example in detail so that someone with little or no web development experience can follow along.

The two source code files, GAS and HTML, run on the server (a Google server somewhere in the world) and the client (the users browser), respectively. This distinction is important because the user’s browser controls the functionality of the form and, since different browsers and browser versions implement HTML5 to varying degrees, the user’s experience will depend on the browser they use. Internet Explorer versions 9 and lower will all be problem-

atic while Chrome and Firefox both have good support for HTML5. The HTML code can also include both JavaScript and CSS. I have used a snippet of JavaScript here that I will explain later but there is no CSS. The client code (HTML + JavaScript + CSS) is all checked by the Google Caja mechanism before it is sent to the client browser. For security reasons, Caja places some limitations on what can be executed; these limits mainly apply to JavaScript. I will discuss the *SandboxMode* option used in the GAS example later but the one used here (*IFRAME*) imposes the fewest restrictions and is the fastest.

The GAS code defines three functions (*onOpen()*, *showForm()* and *getValuesFromForm()*). *onOpen()* is the pre-defined trigger function that builds and adds the menu entitled “Data Entry” as discussed earlier in the chapter.

The *showForm()* function is the one that creates a *HtmlService* object using its *createHtmlOutputFromFile()* method. The method name is self-explanatory; it takes the name of the HTML source file (“index” here minus the “.html” extension) and calls the *setSandboxMode()* method on the returned object. The modes that can be set are defined as an enum and the one chosen here is the *IFRAME* mode. If you do not set a mode, the current default is *NATIVE*. All examples in this chapter use the *IFRAME* because, as described in the [Google documentation](#)<sup>53</sup>, it is the fastest and imposes the fewest restriction. Once the *HtmlService* object is created and its sandbox mode is set, it is displayed by calling the *Spreadsheet* object *show()* method passing the *HtmlService* object as a parameter.

Now that we have a form where we can enter values, we need a mechanism that allows us to retrieve the entered values. The GAS function *getValuesFromForm()* allows us to do this. It takes a *form* object as a parameter and uses the HTML **text input element names** to retrieve the text values that they contain. It then uses the *Spreadsheet* object *appendRow()* method to write the two

---

<sup>53</sup><https://developers.google.com/apps-script/reference/html/sandbox-mode>



entered values as an array to the active sheet. The most interesting part of the whole application -and the crucial part for this simple application- is the manner in which the client-side JavaScript in the HTML source file named “index.html” calls this GAS function. That will be explained below.

There are two parts to the JavaScript code defined in the HTML source file called “index.html”. The shorter piece defines the action for the input button which is to close the form (*google.script.host.close()*) is bound to the *onclick* event of the button. The second piece of JavaScript is a function named *formSubmit()* and it is defined in an HTML script element (between the *<script>* and *</script>* tags). This function is assigned to the *onclick* event of the button labelled “Add Row”. When this button is clicked, the function *formSubmit()* is executed and it, in turn, executes the GAS function *getValuesFromForm()* passing the *form* object itself as an argument. It is important to understand how the JavaScript code in the client interacts with the server-side GAS.

This simple data entry application requires an HTML file. When you create the HTML file, the GAS script editor adds a *<div>* element with opening and closing tags to the file. *<div>* elements are used to group other HTML elements and do not add any visible content to the HTML document. This example does not require *<div>* but a later one will show a common use for them in applying CSS styles to control layout. We create a *<form>* element within the HTML document and add two text boxes and two buttons to it. The text box name attributes are important because they are needed by the GAS code to access the values entered. Finally, the *<br/>* element is used to create line breaks; there are better ways of controlling the layout that will be discussed later. This is a very brief discussion of HTML but for those with little or no background in this topic, there are vast amounts of freely available information on the web. The [W3Schools site](http://www.w3schools.com/html/html_basic.asp)<sup>54</sup> is a good place to start learning.

---

<sup>54</sup>[http://www.w3schools.com/html/html\\_basic.asp](http://www.w3schools.com/html/html_basic.asp)

## 7.3.2 Using HTML Tables to Define Form Layout

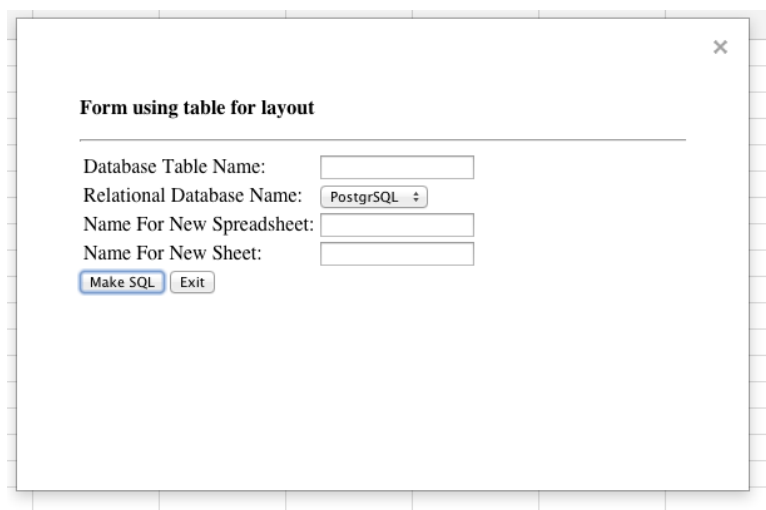
The data entry form discussed above is very minimal in that it uses no CSS and only a few of the available data input HTML elements. In addition, we have accepted all defaults for element size, form size, fonts, and colours. One common trick for defining a data entry form layout is to use HTML tables. While CSS is generally the preferred mechanism for defining layout, HTML tables are often convenient for the types of data input forms used in spreadsheet applications. Code example 7.3 shows how to use an HTML table to create a data input form.

### Code Example 7.3

```
1  <!-- Code Example 7.3 -->
2  <div>
3    <p><b>Form using table for layout</b></p>
4    <hr />
5    <form>
6      <table>
7        <tr>
8          <td>Database Table Name:</td>
9          <td><input id="table_name" name="table_name"
10             type="text" /></td>
11        </tr>
12        <tr>
13          <td>Relational Database Name:</td>
14          <td><select name="rdbms_name"
15             id="rdbms_name">
16             <option value="postgres">PostgreSQL
17             </option>
18             <option value="mysql">MySQL</option>
19             <option value="sqlite">SQLite</option>
20           </select></td>
21        </tr>
```

```
22     <tr>
23       <td>Name For New Spreadsheet:</td>
24       <td><input id="new_spreadsheet_name"
25         name="new_spreadsheet_name" type="text" /></td>
26     </tr>
27     <tr>
28       <td>Name For New Sheet:</td>
29       <td><input id="new_sheet_name"
30         name="new_sheet_name" type="text" /></td>
31     </tr>
32 </table>
33 <input onclick="alert('Clicked button')"
34   type="button" value="Make SQL" />
35 <input onclick="google.script.host.close()"
36   type="button" value="Exit" />
37 </form>
38 </div>
```

In order to view the form, simply copy in the *showForm()* GAS function in code example 7.2 and execute it in the normal manner. Doing this should display the following form:



The screenshot shows a browser window with a form titled "Form using table for layout". The form is organized into a table structure. It contains the following elements:

- Database Table Name:** A text input field.
- Relational Database Name:** A dropdown menu currently showing "PostgreSQL".
- Name For New Spreadsheet:** A text input field.
- Name For New Sheet:** A text input field.
- Buttons:** Two buttons at the bottom, "Make SQL" and "Exit".

Figure 7-3: A form layout created using an HTML table for layout.

If you examine the HTML code used to generate this form, you will see that the text input box labels and the text boxes themselves occur in pairs in table rows (`<tr>` tag) and each is enclosed in a table cell (`<td>` tag). I have also added a drop-down selection in order to introduce the very useful HTML `<select>` element. Personally, I find HTML tables convenient for these types of forms but, as I have stated earlier, using HTML tables for form layout is not regarded as best practice. This form was actually used for an application that I blogged about [here](#)<sup>55</sup>. The form example given above does nothing with the entered values because the button labelled “Make SQL” simply calls the browser `alert()` method rather than calling some GAS function as was done in code example 7.2. If you go to the blog entry link above, you will see how the entered values are used.

Although HTML tables can be convenient for layout, their intended use is to display data in tabular form. CSS is the preferred approach for defining the layout of HTML documents and its use for this is

---

<sup>55</sup><http://www.javascript-spreadsheet-programming.com/2015/04/automated-generation-of-sql-from-google.html>

demonstrated in the next section.

### 7.3.3 Defining Form Layout in CSS

The following code example 7.4 builds a more complicated data entry form and uses CSS to manage the form layout and element colours. Once again we have two source files: one for the GAS and one for the HTML and CSS.

#### Code Example 7.4

```
1  //GAS: code Example 7-4
2  // GAS Code: Example 7-4
3  // Show the web data entry form
4  function showForm() {
5      var ss = SpreadsheetApp.getActiveSpreadsheet(),
6          html = HtmlService
7              .createHtmlOutputFromFile('index')
8              .setSandboxMode(HtmlService
9                  .SandboxMode.IFRAME);
10     html.setWidth(500);
11     html.setHeight(250);
12     ss.show(html);
13 }
14 // Get values from submitted HTML form
15 function getValuesFromForm(form){
16     var firstName = form.firstName,
17         lastName = form.lastName,
18         userAddress = form.userAddress,
19         zipCode = form.zipCode,
20         userPhone = form.userPhone,
21         chosenDate = form.chosenDate,
22         userEmail = form.userEmail,
23         sheet = SpreadsheetApp.getActiveSpreadsheet()
```

```
24         .getActiveSheet();
25     sheet.appendRow([firstName, lastName,
26         userAddress, zipCode,
27         userPhone, chosenDate, userEmail]);
28 }
```

And here is the HTML source file (called “index.html”) that the GAS above uses:

```
1  <!-- Code example 7.4
2  HTML and CSS
3  File name: index.html
4  -->
5  <style>
6      div {
7          background-color: #DCDCDC;
8      }
9      fieldset {
10         display: inline;
11         font-weight: bold;
12     }
13     label {
14         clear: left;
15         float: left;
16         width: 10em;
17         text-align: right;
18         margin-right: 1em;
19         font-weight: bold;
20     }
21     input {
22         background-color: #FFFFFF;
23         width: 14em;
24         float: left;
25     }
```

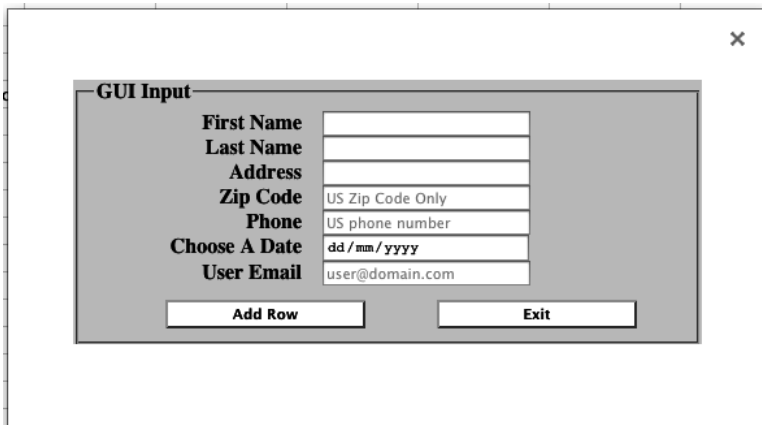
```
26 .button {
27     float: left;
28     margin-left: 5em;
29     background-color: #FFFFFF;
30     color: 000000;
31     margin-top: 1em;
32     font-weight: bold;
33 }
34 </style>
35 <div>
36     <form>
37         <fieldset>
38             <legend>GUI Input</legend>
39             <label>First Name</label>
40             <input type="text" id="first-name"
41                 name="firstName"/>
42             <label>Last Name</label>
43             <input type="text" id="last-name"
44                 name="lastName"/>
45             <label>Address</label>
46             <input type = "text" id = "user-address"
47                 name="userAddress"/>
48             <label>Zip Code</label>
49             <input type=text id="zip-code"
50                 name="zipCode" pattern="(\d{5}([\-]\d{4})?)"
51                 placeholder="US Zip Code Only"/>
52             <label>Phone</label>
53             <input type="text" id="user-phone"
54                 name="userPhone"
55                 placeholder="US phone number"/>
56             <label>Choose A Date</label>
57             <input type="date" id="chosen-date"
58                 name="chosenDate"
59                 placeholder="Pick a date"/>
60             <label>User Email</label>
```

```

61     <input type="email" id="user-email"
62           name="userEmail"
63           placeholder="user@domain.com" required/>
64     <input onclick="formSubmit()" type="button"
65           value="Add Row" class="button"/>
66     <input onclick="google.script.host.close()"
67           type="button"
68           value="Exit" class="button"/>
69   </fieldset>
70 </form>
71 </div>
72 <script type="text/javascript">
73   function formSubmit() {
74     google.script.run
75       .getValuesFromForm(document.forms[0]);
76   }
77 </script>

```

Executing the GAS function *showForm()* should display a form like the one shown in figure 7-4 below.



The screenshot shows a browser window with a form titled "GUI Input". The form contains the following fields and buttons:

<b>First Name</b>	<input type="text"/>
<b>Last Name</b>	<input type="text"/>
<b>Address</b>	<input type="text"/>
<b>Zip Code</b>	US Zip Code Only
<b>Phone</b>	US phone number
<b>Choose A Date</b>	dd/mm/yyyy
<b>User Email</b>	user@domain.com

At the bottom of the form, there are two buttons: "Add Row" and "Exit".

Figure 7-4: A Form created using CSS.



The code used to generate the form shown in figure 7-4 introduces some new and useful HTML features in addition to its use of CSS to control colours and the layout. It uses the `<fieldset>` HTML element to contain the data entry labels and text inputs. Some of the text input elements, for example the input labelled “Zip Code”, have set a placeholder attribute to show greyed-out text as a hint for what the user should enter. The form also uses different input types from just plain text. The *date* and *email* inputs use types of **email** and **date**, respectively. These are HTML5 additions and the date one particularly interesting. If you are using a browser that supports the date type, then a very useful calendar will appear when the users enters the cursor in the input with the date type. As of time of writing, Chrome supports this but Firefox does not but you can also get the same calendar display using jQuery. However, although Firefox does not support it, it still allows the user to enter date values as string. I am not going to discuss these HTML5 enhancements any further here but it is well worth being aware of them and using them where appropriate.

The main point of this section is the use of CSS. The CSS code is defined in the `<style>` element outside the HTML enclosed by the `<div>` element. CSS is a large topic and in-depth discussion of it is beyond the scope of this book. As with HTML, there is a lot of freely available material devoted to it on the web and the [W3Schools CSS](http://www.w3schools.com/css/)<sup>56</sup> section is a good place to start exploring it in more detail. In brief, CSS allows us to select and apply styles to any HTML element based on its element type, attribute name or id value. Styles can be selectively applied to elements only when they are contained in other elements; the `<div>` element discussed briefly earlier in the chapter is sometimes used for this. To learn more about CSS and how it can be used to style *HtmlService* forms, I suggest reading a basic CSS tutorial and then playing with the code example given above by changing the CSS, saving the source code, and then displaying the form by executing the GAS `showForm()` function.

---

<sup>56</sup><http://www.w3schools.com/css/>

## 7.4 Transferring Data from Google Sheets To an *HtmlService* Web Application

The preceding examples showed how to transfer data from a data input form to a Google Sheet. Sometimes we may wish to do the reverse and take data from spreadsheet cells and display that data in our web application. I will discuss two uses of this approach. The first uses the *HtmlService* templating ability to take cell values from the spreadsheet and display them in an HTML table in the web application. This type of dynamic HTML generation is analogous to how PHP and ASP work. The second set of examples that I will show are, I believe more useful; they take spreadsheet cell values and use them to dynamically populate HTML elements like the `<select>` drop-down.

### 7.4.1 Templating

*HtmlService* supports tags that can be embedded in the HTML file that can then be evaluated to dynamically fill in values. The code below adds some dummy data to a Google Sheet, gives the created data a range name and then uses the range name to retrieve the data and finally loops over the returned arrays data structure to dynamically create an HTML table that is displayed in a sidebar.

#### Code Example 7.5

```
1 // Add some dummy data to the active sheet.
2 // Data from:
3 // "http://www.w3schools.com/html/html_tables.asp"
4 function addRows() {
5     var ss = SpreadsheetApp.getActiveSpreadsheet(),
6         sheet = ss.getActiveSheet(),
7         rows = [['Number', 'First Name',
8                 'Last Name', 'Points'],
9                 [1, 'Eve', 'Jackson', 94],
10                [2, 'John', 'Doe', 80],
11                [3, 'Adam', 'Johnson', 67],
12                [4, 'Jill', 'Smith', 50]],
13     rng,
14     rngName = 'Input';
15     rows.forEach(function(row) {
16         sheet.appendRow(row);
17     });
18     rng = sheet.getDataRange();
19     ss.setNamedRange(rngName, rng);
20 }
21
22 // Take the template file and fill in the data
23 // in range named "Input".
24 // Display the data as a sidebar.
25 function displayDataAsSidebar() {
26     var html = HtmlService
27         .createTemplateFromFile('DummyData');
28     SpreadsheetApp.getUi()
29         .showSidebar(html.evaluate());
30 }
31 // Return the data in the range named "Input"
32 // as an array-of arrays.
33 function getData(){
34     var ss = SpreadsheetApp.getActiveSpreadsheet(),
35         rng = ss.getRangeByName('Input'),
```

```

36   data = rng.getValues();
37   return data;
38 }

```

The HTML template file code:

```

1  <!-- Code Example 7-5
2  File name: DummyData.html -->
3  <style>
4      table {
5          border-collapse: collapse;
6      }
7      td {
8          border: 1px solid black;
9      }
10 </style>
11
12 <div> <h1>Dummy Data</h1>
13 <? var data=getData(); ?>
14 <table>
15     <? for(var i=0;i<data.length;i++){?>
16         <tr>
17             <? for(var j = 0;j < data[i].length;
18                 j++) { ?>
19                 <td><?= data[i][j] ?></td> <? } ?>
20         </tr>
21     <? } ?>
22 </table>
23 </div>

```

The functions *getData()* and *addRows()* use standard GAS objects and methods to add data to the sheet, give the data range a name and return the data as an array of arrays. Once again, our application contains two source files: the GAS file and the HTML file. The

difference is that this time the HTML file is a template that uses special tags `<??>` and `<?=??>`. Valid GAS statements can be run within the `<??>` tag and value assignments can be made with the `<?=??>` tag. Any GAS variables assigned within the template code are then accessible to subsequent GAS code. For example, the variable `data` is assigned to the nested array returned by the GAS function `getData()` in the template line `<? var data=getData(); ?>` and the nested array is then used to populate the HTML table cells in the nested for loop that follows. The novel part of the GAS code is the dynamic evaluation of the template file in the function `displayDataAsSidebar()` where the `HtmlService` method `createTemplateFromFile()` takes the template file name minus the “.html” extension as an argument. The `Ui` object is created by the method call `SpreadsheetApp.getUi()` and its `showSidebar()` method is called with the evaluated template object as its argument. After calling the `displayDataAsSidebar()` function, you should see something like figure 7-5 in your spreadsheet view.

Number										
A	B	C	D	E	F	G	H	I		
Number	First Name	Last Name	Points							
1	Eve	Jackson	94							
2	John	Doe	80							
3	Adam	Johnson	67							
4	Jill	Smith	50							

Dummy Data			
Number	First Name	Last Name	Points
1	Eve	Jackson	94
2	John	Doe	80
3	Adam	Johnson	67
4	Jill	Smith	50

Figure 7-5: Output from a template shown as a sidebar on right-hand side.

Templates could be put to more practical use than shown here. One use could be for displaying dynamic summaries of data but the code shown demonstrates the general technique.

## 7.4.2 Dynamically Populating HTML Elements with Values Stored in Sheet Ranges

In Excel VBA it is quite easy to transfer values from spreadsheet cells to form controls such as listboxes and comboboxes. This is a useful feature because the developer can hide and protect the ranges that provide the values and keep them separate from the

code that uses them. The code in example 7.6 shows how to do this in GAS by populating an HTML `<select>` element from values stored in a Sheets range. The code employs some useful GAS tricks for dynamically re-assigning a name to a range as the rows are added. In addition, it demonstrates how client-side JavaScript can interact with GAS and use values returned by GAS functions.

### Code Example 7.6

```
1 // Display the GUI
2 // Execute this function from the script editor oo run
3 // the application.
4 // Note the call to "setRngName()".
5 // This ensures that all newly added
6 // values are included in the dropdown
7 // list when the GUI is displayed
8 function displayGUI() {
9     var ss,
10         html;
11     setRngName();
12     ss = SpreadsheetApp.getActiveSpreadsheet();
13     html = HtmlService.createHtmlOutputFromFile('index')
14         .setSandboxMode(HtmlService.SandboxMode.IFRAME);
15     ss.show(html);
16 }
17 // Called by Client-side JavaScript in the index.html.
18 // Uses the range name argument to extract the values.
19 // The values are then sorted and returned as an array
20 // of arrays.
21 function getValuesForRngName(rngName) {
22     var rngValues = SpreadsheetApp.getActiveSpreadsheet()
23         .getRangeByName(rngName).getValues();
24     return rngValues.sort();
25 }
26 //Expand the range defined by the name as rows are added
```

```

27 function setRngName() {
28   var ss = SpreadsheetApp.getActiveSpreadsheet(),
29       sh = ss.getActiveSheet(),
30       firstCellAddr = 'A2',
31       dataRngRowCount = sh.getDataRange().getLastRow(),
32       listRngAddr = (firstCellAddr +
33                     ':A' + dataRngRowCount),
34       listRng = sh.getRange(listRngAddr);
35   ss.setNamedRange('Cities', listRng);
36 }
37
38 // Return the data in the range named "Input"
39 // as an array-of arrays.
40 function getData(){
41   var ss = SpreadsheetApp.getActiveSpreadsheet(),
42       rng = ss.getRangeByName('Input'),
43       data = rng.getValues();
44   return data;
45 }

```

The accompanying HTML source code:

```

1  <!--
2  Code Example 7-6
3  File Name index.html
4  Creates a GUI that gets the dropdown list elements from\
5  a Google Spreadsheet
6  named range. As values are added to, the range name is \
7  re-assigned to include
8  the added values.
9  -->
10 <p>Getting Dropdown Values From Spreadsheet
11   Dynamically </p>
12 <hr>

```

```
13 <div>
14   <form>
15     <table>
16       <tr>
17         <td>Select A City</td><td><select id="city_list"
18           ></select></td>
19       </tr>
20       <tr>
21         <td>Add other form elements here
22           </td><td>
23         <input onclick="google.script.host.close()"
24           type="button" value="Exit" /></td>
25       </tr>
26     </table>
27   </form>
28 </div>
29 <script type="text/javascript">
30   // Client-side JavaScript that uses the list return\
31 ed by
32   // GAS function "getValuesForRngName()" to populate\
33 the dropdown.
34   // This is standard client-side JavaScript programm\
35 ing that uses
36   // DOM manipulation to get page elements and manipu\
37 late them.
38   function onSuccess(values) {
39     var opt,
40       dropDown;
41     for(i = 0; i < values.length; i +=1){
42       dropDown = document.getElementById("city_list\
43 ");
44       opt = document.createElement("option");
45       dropDown.options.add(opt);
46       // Remember that GAS Range method
47       // "GetValues()" returns an array
```



```

48         // of arrays, hence two array
49         //   indexes "[i][0]" here.
50         opt.text = values[i][0];
51         opt.value = values[i][0];
52     }
53 }
54 function populate(){
55     google.script.run.withSuccessHandler(onSuccess)
56         .getValuesForRngName('Cities');
57 }
58 </script>
59 <script>
60     // Using the "load" event to execute the function "po\
61     pulate"
62     window.addEventListener('load', populate);
63 </script>

```

If you add values to column A of the active sheet and then execute the function *displayGUI()*, you should see something like the screenshot shown in figure 7-6 below.

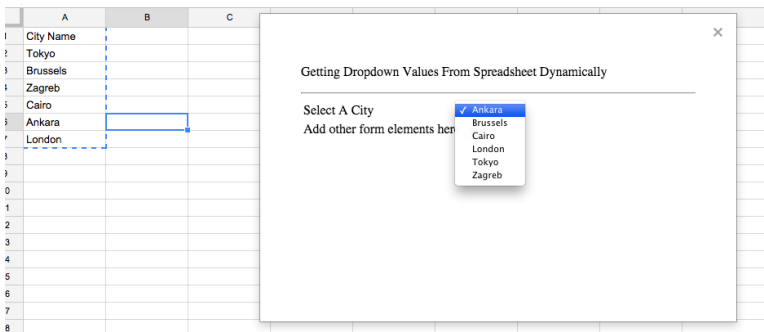


Figure 7-6: Values in a named spreadsheet range populating a form dropdown list.

The above code demonstrates how we can add new values to the list in column A of the active sheet that are then picked up each time

the form is displayed. The key to how it all works is the ability of client-side JavaScript embedded in the HTML file to execute GAS on the Google server and use the GAS function return value. This code example is the most complex example in this chapter so I will give a detailed explanation of how it works. When the GAS function *displayGUI()* is executed, it first calls another GAS function, *setRngName()*, that sets the name “Cities” for all cells in column A, starting with and including cell A2, that contain values. The remainder of the code in *displayGUI()* is composed of *HtmlService* method calls that create the GUI from the HTML file called “index.html”. When the GUI loads, it fires an event called “load” and this event causes the client-side JavaScript function called *populate()* to fire. The line *window.addEventListener('load', populate);* is at the bottom of the “index.htm”l file. The function *populate()* is the part of the code that ties the GAS and client-side JavaScript together:

```
1  function populate(){
2    google.script.run.withSuccessHandler(onSuccess)
3      .getValuesForRngName('Cities');
4  }
```

This function executes the GAS function *getValuesForRngName()* passing it a hard-coded string argument “Cities” that is the name defined for the spreadsheet range that contains the values we want to transfer to the GUI drop-down. We need to define another client-side JavaScript function to execute when the call to the GAS function succeeds. The function that is passed in is called *onSuccess()* in this example. The *onSuccess()* function receives the array of arrays returned by the GAS function *getValuesForRngName()*. It uses the given argument values to add options to the `<select>` element. The code here is straightforward client-side JavaScript code that uses Document Object Model (DOM) methods to get and manipulate DOM elements (*document.getElementById()* and *document.createElement()*).

We can use the same technique of taking spreadsheet range values to dynamically create other HTML elements. For example, suppose we want to create a set of radio buttons to allow exclusive selection from a group of options, we can do this by replacing the *onSuccess()* JavaScript function definition in the HTML file with the following:

```
1     function onSuccess(values) {
2         var radioInput,
3           form = document
4             .getElementById('test_btn_add'),
5           label;
6         for(var i = 0; i < values.length; i +=1){
7             //alert(values[i][0]);
8             label = document.createElement("label");
9             radioInput = document.createElement('input');
10            radioInput.setAttribute('type', 'radio');
11            radioInput.setAttribute('name', 'cities');
12            radioInput.setAttribute('id', 'cities');
13            radioInput.setAttribute('value',
14                values[i][0]);
15            label.appendChild(radioInput);
16            label.innerHTML += values[i][0];
17            form.appendChild(label);
18        }
19    }
```

Now when we execute the GAS *displayGUI()* function, we should see the form as shown in figure 7-7 below.

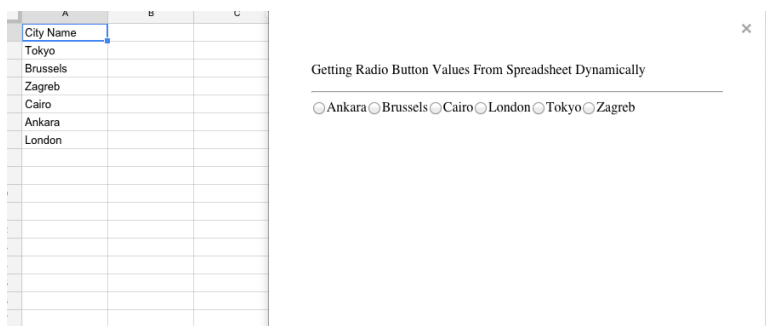


Figure 7-7: Dynamically creating a set of radio buttons using spreadsheet range values.

Readers unfamiliar with web development might find the JavaScript in the `onSuccess()` function difficult. The DOM methods are used to select HTML elements (`getElementById()`), create elements (`createElement()`) and set attribute elements (`setAttribute()`). An in-depth discussion of the DOM is beyond the scope of this book but, once again, there are ample freely available and excellent resources available on the web for learning about it. An excellent place to start is the [Mozilla Developer Networks tutorial](#)<sup>57</sup>.

## 7.5 Create Professional-looking Forms the Easy Way - Use Bootstrap

By using HTML tables or, better still, by using CSS, we can create usable forms that do not look so bad. However, we can take a short cut and avail of some excellent and publicly available code to do all the CSS for us. [Bootstrap](#)<sup>58</sup> is a publicly available library of HTML and CSS made available by Twitter that we can use for developing

<sup>57</sup>[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Introduction](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

<sup>58</sup>[https://en.wikipedia.org/wiki/Bootstrap\\_%28front-end\\_framework%29](https://en.wikipedia.org/wiki/Bootstrap_%28front-end_framework%29)

web applications. “It is the most-popular project on GitHub by far.”<sup>59</sup> I will briefly introduce how we can take advantage of Bootstrap in GAS. Here is a data registration form created with the help of Bootstrap:

Figure 7-8: Bootstrap data entry form.

### Code Example 7.7

```

1  <!--
2  Code Example 7.7
3  Form code taken from http://www.tutorialspoint.com/boot\
4  strap/bootstrap_forms.htm
5  -->
6  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/
7      3.3.4/css/bootstrap.min.css" rel="stylesheet"
8  t">
9  <div>
10 <form class="form-horizontal" role="form">

```

<sup>59</sup><http://www.infoworld.com/article/2606567/application-development/120001-GitHub-s-top-10-rock-star-projects.html#slide2>

```
11     <div class="form-group">
12         <label for="firstname" class="col-sm-2
13             control-label">First Name</label>
14         <div class="col-sm-10">
15             <input type="text" class="form-control"
16                 id="firstname"
17                 placeholder="Enter First Name">
18         </div>
19     </div>
20     <div class="form-group">
21         <label for="lastname" class="col-sm-2
22             control-label">Last Name</label>
23         <div class="col-sm-10">
24             <input type="text" class="form-control"
25                 id="lastname"
26                 placeholder="Enter Last Name">
27         </div>
28     </div>
29     <div class="form-group">
30         <div class="col-sm-offset-2 col-sm-10">
31             <div class="checkbox">
32                 <label>
33                     <input type="checkbox"> Remember me
34                 </label>
35             </div>
36         </div>
37     </div>
38     <div class="form-group">
39         <div class="col-sm-offset-2 col-sm-10">
40             <button type="submit" class="btn btn-default">
41                 Sign in</button>
42         </div>
43     </div>
44 </form>
45 </div>
```

GAS code to display the form:

```
1 // Display a GUI that uses Bootstrap to do the form styling
2
3 function displayGUI() {
4     var ss,
5         html;
6     ss = SpreadsheetApp.getActiveSpreadsheet();
7     html = HtmlService
8         .createHtmlOutputFromFile('index')
9         .setSandboxMode(HtmlService
10            .SandboxMode.IFRAME);
11     ss.show(html);
12 }
```

Executing the GAS function *displayGUI()* above should display the form shown in figure 7-7 above. I am not going to discuss Bootstrap in detail other than to highlight two aspects of the HTML code given above. Firstly, the Bootstrap CSS library is accessed in the `<link>` element at the top of the HTML file and the Bootstrap file is provided by a Content Delivery Network (CDN). The second point concerns how the Bootstrap CSS code applies the styles to the HTML elements. Bootstrap uses the element class attributes to determine the style to apply. For example note the code `class="form-horizontal"` in the `<form>` element. By using a set of Bootstrap-defined class attribute names for our elements, we can control what styles Bootstrap will apply to those elements.

This is a very brief introduction to Bootstrap. For those who would like to take this further, there are abundant examples available on the web. W3Schools has a good introduction to [Bootstrap that is worth reading](http://www.w3schools.com/bootstrap/).<sup>60</sup> Another very commonly used resource in web development is the JavaScript JQuery library. It can also be used

---

<sup>60</sup><http://www.w3schools.com/bootstrap/>

in GAS via a CDN. Since it is JavaScript, you reference it using the `<script>` element.

## 7.6 Summary

End-users of spreadsheet applications generally expect some sort of GUI. Google Sheets allows developers to provide menus and web applications. We can use the trigger function `onOpen()` to define code to add a menu to the menu bar. Web applications are developed using *HtmlService*. *HtmlService* development requires some knowledge of HTML, CSS and client-side JavaScript. We have seen how to develop simple data entry forms and how we can use both HTML tables and CSS to control the form layout. *HtmlService* provides a templating mechanism that we can use to dynamically generate HTML. We have also seen how we can use ranges stored in spreadsheet cells to populate form drop-downs and radio button options when the form is loaded. These examples also showed how to call GAS functions from client-side JavaScript. Bootstrap was introduced to show how it can be used to create professional-looking web applications. *HtmlService* offers the full power of HTML, CSS and client-side JavaScript to create very sophisticated web applications. It would require a book of its own to do justice to what is possible but, hopefully, this chapter provides a good introduction to get readers started.



# Chapter 8: Google Drive, Folders, Files, And Permissions

## 8.1 Introduction

This and the following chapter mark a switch in emphasis from covering GAS specifically for spreadsheet manipulation to using GAS to control Google Drive and Gmail. The GAS scripts will still be written and executed in the Sheets Script editor but they will be used to manipulate entities such as files and folders and emails. Google Drive and Gmail are two important components of the Google suite of tools and they complement Google Sheets very well. Many spreadsheet applications need to create files and folders, set permissions and send emails. All these tasks can be accomplished using GAS hosted in Sheets. I cover the creation of files and folders and the setting of their permissions in this chapter. Gmail is covered in the next chapter.

The Google Drive that comes with each Gmail account is analogous to a file system on a Windows or Mac desktop. There are some important differences, however, that will be pointed out as they become relevant to the examples given. A Google Drive can contain a variety of uploaded file types from PDFs to various image file types in addition to the Google Sheets, documents, and presentation files. As with standard desktop computers, you can create folders, add files to these folders, remove files from them and create folders within folders. All of this manipulation can be performed in GAS using the *DriveApp* object.

The previous version of this book used the now discontinued *Doc-*

*sList* object. Code examples that use this object no longer work and have to be migrated to use *DriveApp* instead. *DriveApp* implements the same functionality as its predecessor *DocsList* but does it in a different way. For example, where *DocsList* methods returned arrays, *DriveApp* typically returns an **iterator** as will be explained below. Although *DocsList* is no more, you will still find examples on older Stackoverflow postings that use it but this code will not work.



## ***DocsList* Is Gone!**

*DocsList* has been replaced by *DriveApp*.

This chapter covers the *DriveApp*<sup>61</sup> object in some detail using practical examples to demonstrate the key concepts. The *DriveApp* object implements a comprehensive set of methods that allows you to list files and folders, create and remove files and folders, move files to and from folders and set file permissions. Since you can perform almost any file operation using *DriveApp* methods, you need to be careful when writing GAS scripts or when running GAS scripts provided by others that use *DriveApp* methods. Otherwise, you could end up losing files and data.

This chapter covers the basics of manipulating files and folders. It shows how to create and remove them, add files to folders, and add viewers and editors to files and folders to facilitate collaborative working. It closes with practical examples that provide code to consolidate the knowledge gained and to perform some common tasks.

This chapter concentrates on methods of objects in the *DriveApp* hierarchy. If you have studied the examples in preceding chapters, you should already know how to read spreadsheet cell values into GAS data structures (arrays and objects) and how to write data back from these data structures to spreadsheet cells. I will, however, use

---

<sup>61</sup><https://developers.google.com/apps-script/reference/drive/drive-app>

the examples in this chapter to explore some useful programming topics such as recursion that have not been covered so far.

All the GAS code for this chapter, and there is quite a lot of it, is available on [GitHub](#)<sup>62</sup>

## 8.2 List Google Drive File And Folder Names

A user's Google Drive can contain one or more folders that can be created manually from the Google Drive interface or programmatically by GAS using the *DriveApp* object. Listing all the folders in a Google Drive is quite straightforward and will serve as a starting point for exploring the *DriveApp* object. The following function performs this operation and writes the folder names to the logger.

### Code Example 8.1

```
1 // Write user's Google Drive folder names to
2 // the logger.
3 function listDriveFolders() {
4     var it = DriveApp.getFolders(),
5         folder;
6     Logger.log(it);
7     while(it.hasNext()) {
8         folder = it.next();
9         Logger.log(folder.getName());
10    }
11 }
```

In the above example we use the *DriveApp* *getFolders()* method to return an **iterator**<sup>63</sup>. An iterator is an object that provides methods

---

<sup>62</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015/blob/master/ch08.gs](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch08.gs)

<sup>63</sup><https://en.wikipedia.org/wiki/Iterator>

to process a collection of elements without having to read the entire collection into memory as is done when we use an array. In chapter 6 we encountered cursors which are also a type of iterator where we access the rows returned from the database by moving the cursor forward through the record set. Instead of returning an array of elements, the method *getFolders()* returns an iterator, hence the variable name *it*, and this iterator provides the methods to check if we have reached the end of the iterator (*hasNext()*) and to extract a value from the iterator (*next()*). The *DriverApp* object's predecessor, the *DocsList* object, had a *getAllFolders()* method which returned an array of folder objects. Using an array-based approach is fine when you know that the number of returned elements poses no memory issues but will cause problems when the array size exceeds available memory. Perhaps this is why Google replaced array-returning methods with the iterator approach. In this example we have simply printed the folder names to the log.

The line *Logger.log(it);* in example 8.1 prints **FolderIterator** to emphasise the iterator nature of the object that we use to traverse the folder names. The *hasNext()* method returns *false* when there are no more objects to return, that is in iterator parlance when we have **exhausted** the iterator. If *hasNext()* returns *true*, we know we have not yet exhausted the iterator so we can call its *next()* method to return another object for processing. *DriveApp* uses iterators extensively so it is important to understand the concept of a method that returns an iterator of some kind, using the *hasNext()* method to check if there are still elements to process and then accessing those elements using the iterator's *next()* method.

Once you understand the concept of an iterator, this code example is quite straightforward. In a real application we could write the folder names to a sheet or return an array of *Folder* objects for further manipulation by other GAS functions. The *getFolders()* method used in this example returns **all** folders including sub-folders.

Exactly the same approach can be used to get a file listing as shown

in example 8.2.

### Code Example 8.2

```
1 // Write the names of files in the user's
2 // Google Drive to the logger.
3 function listDriveFiles() {
4     var fileIt = DriveApp.getFiles(),
5         file;
6     Logger.log("File Names:");
7     while (fileIt.hasNext()) {
8         file = fileIt.next()
9         Logger.log(file.getName());
10    }
11 }
```

In this example we create a *FileIterator* (*filesIt*) in place of the *FolderIterator* created earlier. Otherwise, the methods and their use is exactly the same as in the folder example. As with the folder example, this example also prints names for all files regardless of where they are in the folder hierarchy.

There is always one folder named “My Drive” in every Google Drive. However, this folder name is not printed by the folder-listing code given earlier. It is the top-level folder and can be regarded as the containing folder for all user-created files and folders. It is treated as the **root** folder and can be accessed in GAS as shown in the following example:

### Code example 8.3

```
1 // Display the name of the top-level
2 // Google Drive folder as a message box.
3 function showRoot() {
4     var root = DriveApp.getRootFolder();
5     Browser.msgBox(root.getName());
6 }
```

We will use the method `getRootFolder()` to return a reference to the “Root” folder in later examples.

## 8.3 Creating And Removing Files And Folders

Creating files and folders is not difficult but something to be aware of is that **there can be duplicate file names or folder names in the same folder**. This is an important difference from conventional file systems.

### Code Example 8.4

```
1 // Demonstration code only to show how
2 // Google Drive allows duplicate file
3 // names and duplicate folder names
4 // within the same folder.
5 function makeDuplicateFilesAndFolders() {
6     SpreadsheetApp.create('duplicate spreadsheet');
7     SpreadsheetApp.create('duplicate spreadsheet');
8     DriveApp.createFolder('duplicate folder');
9     DriveApp.createFolder('duplicate folder');
10 }
```

After executing the code above, the duplicate files and folders should be visible in the Google Drive view. So how can such

duplicates be distinguished? The answer is that they **each have different identifiers (ID)**. These identifiers appear in the URL when either the file or folder is opened. For folders, the URL ends in “folders/” followed by a long alphanumeric string and for the spreadsheet files there will be something like “key=” followed by the long alphanumeric string. The important point is that **Google Drive uses the ID and not the file or folder name** to uniquely identify these objects. The following code example shows how folders with the same name located in the same parent folder (Root in this instance) have different IDs.

### Code Example 8.5

```

1  // Code Example 8.5
2  // Demonstrates how folders can have the
3  // same name and parent folder
4  // (Root in this instance) but yet
5  // have different IDs.
6  function writeFolderNamesAndIds() {
7      var folderIt = DriveApp.getFolders(),
8          folder;
9      while(folderIt.hasNext()) {
10         folder = folderIt.next();
11         Logger.log('Folder Name: ' +
12                 folder.getName() +
13                 ', ' +
14                 'Folder ID: ' +
15                 folder.getId());
16     }
17 }

```

After executing code example 8.5, the duplicate folders created in example 8.4 reveal something interesting. Here is my logger output:

```

[15-09-22 08:35:31:306 BST] Folder Name: duplicate folder, Folder
ID: 0B2dsdq7IKB9yem52cUlmZXV5X1k [15-09-22 08:35:31:308 BST]

```

*Folder Name: duplicate folder, Folder ID: 0B2dsdq7IKB9yTW1YWVczQV9CU2c*

I have two folders with the same name -“duplicate folder”- but with different IDs. That is the important point: Names and folder locations do not confer uniqueness in Google Drive, it is the IDs that do so.

Having produced the useless duplicated spreadsheet files and folders, we can now remove them. The following function does just that. It uses string literals to identify the target files and folders and it then removes them.

### Code Example 8.6

```

1  // Remove any folders with the given argument name.
2  // Caution: Make sure no required
3  function removeFolder(folderName) {
4      var folderIt =
5          DriveApp.getFoldersByName(folderName),
6          folder;
7      while(folderIt.hasNext()) {
8          folder = folderIt.next();
9          folder.setTrashed(true);
10     }
11 }
12 // Call "removeFolder()" passing the name of the
13 // dummy folders created in example 8.4 above.
14 function removeTestDuplicateFolders() {
15     var folderName = 'duplicate folder';
16     removeFolder(folderName);
17 }
18 // Remove all files called 'duplicate spreadsheet'
19 // created in example 8.4 above.
20 function removeDummyFiles() {
21     var fileIt =
22         DriveApp.GetFilesByName('duplicate spreadsheet'),

```



```
23     file;
24     while(fileIt.hasNext()) {
25         file = fileIt.next();
26         file.setTrashed(true);
27     }
28 }
```

The duplicated folders and files were all placed in the top-level Google Drive default folder “My Drive”. We use the *getFoldersByName()* and *getFilesByName()* methods to respectively identify the folders and files to be removed. These methods once again return iterators that are accessed using the same *hasNext()* and *next()* methods discussed previously. We then call the *setTrashed()* method on the file or folder object returned by the iterator *next()* method passing it the Boolean *true*.

Armed with knowledge of how to create and use file and folder iterators to list files and folders as well as to create and delete them, we can move on to more interesting examples.

## 8.4 Adding Files To And Removing Files From Folders

In a conventional file share folders and sub-folders are a standard mechanism for organising files. They can be used in the same way in Google Drive but, as with duplicate names, there are some differences. The following function creates a new file, a new folder and then adds the file to the folder:

### Code Example 8.7

```
1 // Create a test folder and a test file.
2 // Add the test file to the test folder.
3 function addNewFileToNewFolder() {
4     var newFolder =
5         DriveApp.createFolder('Test Folder'),
6         newSpreadsheet =
7             SpreadsheetApp.create('Test File'),
8         newSpreadsheetId = newSpreadsheet.getId(),
9         newFile =
10            DriveApp.getFileById(newSpreadsheetId);
11     newFolder.addFile(newFile)
12 }
```

When the function `addNewFileToNewFolder()` is executed, a new folder called “Test Folder” and a new spreadsheet file called “Test File” will appear in Google Drive.

You should note that the *Spreadsheet* object returned by the *SpreadsheetApp create()* method is not an object that is recognised by *DriveApp* methods. To obtain an object that *DriveApp* methods recognise, we must first get the *Spreadsheet* file ID using its *getId()* method. We can then pass this ID to the *DriveApp getFileById()* method to return a *File* object. This may appear unnecessarily convoluted but *DriveApp* has to deal with files of many types and not just Google Sheets so it only recognises a generic *File* object.

Although the new spreadsheet file appears in both the root and in the newly created folder, it is not duplicated. If it is deleted from one location, it will also disappear from the other. In order to tidy things up, the newly created file can be removed from the root folder as follows:

### Code Example 8.8

```
1 // Remove the file from root folder.
2 // Does not delete it!
3 // ID was taken from the URL.
4 // Warning: Replace the ID below with your own
5 // or you will get an error.
6 function removeTestFileFromRootFolder() {
7     var root = DriveApp.getRootFolder(),
8         id =
9         '1xIeidH1-Em-xD1m_84e70FrORfkttXFvVqYRzEv58ZA',
10        file = DriveApp.getFileById(id);
11    root.removeFile(file);
12 }
```

The ID used to identify the test file was taken from the URL. When running this code, you will need to extract the ID from the URL for the file created in the earlier code example and replace the one given here with your own for this code to run. The important point to note here is that, although the file was removed from the root folder, it still appears in the newly created test folder.

File folder membership can be ascertained using the *File* object *getParents()* method which returns a *FolderIterator* object. If a file has not been added to any folder, then this *getParents()* method will simply return an iterator with one *Folder* object named “My Drive”. Code example 8.9 below contains two functions. The first creates a folder and a spreadsheet file that is then added to the new folder and returns the ID of the new file. The second function uses this ID to get a *File* object using the *DriveApp* *getFileById()* method. The *File* object method *getParents()* to return a *FolderIterator* object that is then traversed using the methods discussed earlier.

### Code Example 8.9

```
1 // Demonstrate how a file can have multiple parents.
2 //
3 // Create a folder and a spreadsheet.
4 // Add the newly created spreadsheet to the new folder
5 // and return the ID of the new spreadsheet.
6 // Used by function "getFileParents()" below.
7 function createSheetAndAddToFolder() {
8     var newFolder =
9         DriveApp.createFolder('TestParent'),
10        gSheet = SpreadsheetApp.create('DummySheet'),
11        gSheetId = gSheet.getId(),
12        newFile = DriveApp.getFileById(gSheetId);
13    newFolder.addFile(newFile);
14    return gSheetId;
15 }
16 // Call the function above to create a folder and
17 // spreadsheet and add the spreadsheet to the file.
18 // Get the file object using the returned ID and
19 // call its "getParents()" method. Traverse the
20 // returned iterator to print out the parent names.
21 function getFileParents() {
22     var gSheetId = createSheetAndAddToFolder(),
23         file = DriveApp.getFileById(gSheetId),
24         gSheetParentsIt = file.getParents(),
25         parentFolder;
26     while(gSheetParentsIt.hasNext()) {
27         parentFolder = gSheetParentsIt.next();
28         Logger.log(parentFolder.getName());
29     }
30 }
```

If you include the `getParents()` method call in the same function that creates the folder and file and adds the file to the folder, the new folder is not listed as a parent. It seems that the parents are not updated within the scope of the function but breaking the code into

two functions as done above correctly reports the parents as “My Drive” and “TestParent” in the log.

The main point of code example 8.5 is to demonstrate that in Google Drive, unlike in conventional file systems, **the same file can belong to more than one folder.**

Files and folders have many more methods than I have covered here. You can see the full list by creating a *File* or *Folder* object in the GAS script editor and then see the auto-members list that the editor displays after the period (.). Alternatively, you can use the JavaScript *Object.keys()* method as described in section 8.4 on the *Sheet* object to get an array of properties that you can then print to the logger or to a Google Sheet. At the time of writing the *Folder* and *File* objects have 45 and 42 properties (mostly methods), respectively.

## 8.5 File And Folder Permissions

Google documents are very much designed with collaboration in mind. Any Gmail account holder can create a Google Sheet, for example, and then allow other Gmail account users to view and, if desired, to edit the document. Files and folders can be shared using the dialog displayed by right-clicking the mouse when the target file or folder is selected. For a file, this brings up the dialog shown in figure 8-1 below.

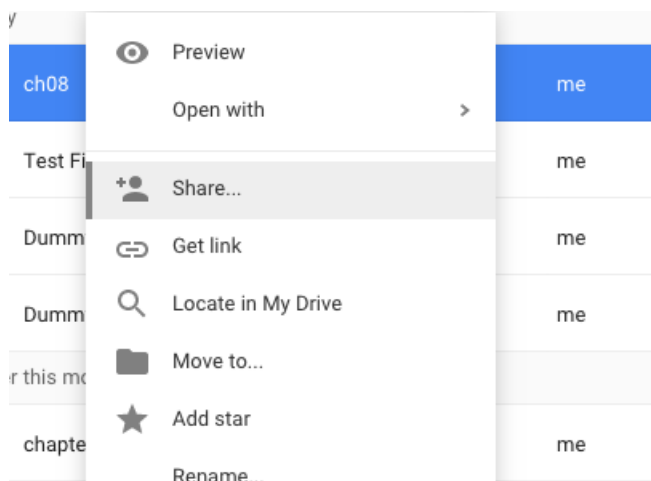


Figure 8-1: File sharing dialog in Google Drive

Since adding viewers and editors to files and folders can be done from the Google Drive interface, you might ask why GAS is required. The main reason is that GAS can be used to automate the process and can allow very fine-grained control. In addition, lists of viewers and editors can be managed in Google Sheets and GAS can easily manipulate and utilise these lists. In order to follow along with the examples given below, you will need another Gmail account or the cooperation of another account holder in order to see the effects of one user adding viewer/editor permissions to folders and files for another user. For the examples here, I use my “*mick@javascript-spreadsheet-programming.com*” account with my personal account with the latter creating the files and folders and then granting or revoking permissions for the former.

### 8.5.1 Creating The Files and Recording The Relevant Details

Code example 8.10 demonstrates file permissions for two newly-created empty, files.

**Code Example 8.10**

```
1 // Creates two files in "My Drive".
2 // (1) A spreadsheet and (2) A document.
3 // Records the name, ID and URL for the
4 // two new files in the active spreadsheet.
5 // Gives a range name to the two data rows.
6 // Script requires authorisation.
7 function createFiles() {
8     var ss = SpreadsheetApp.getActiveSpreadsheet(),
9         sh = ss.getActiveSheet(),
10        newSsName = 'myspreadsheet',
11        newSs = SpreadsheetApp.create(newSsName),
12        newDocName = 'mydocument',
13        newDoc = DocumentApp.create(newDocName);
14    sh.appendRow(['File Name',
15                'File ID',
16                'File URL']);
17    sh.getRange(1, 1, 1, 3).setFontWeight('bold');
18    sh.appendRow([newSsName,
19                 newSs.getId(),
20                 newSs.getUrl()]);
21    sh.appendRow([newDocName,
22                 newDoc.getId(),
23                 newDoc.getUrl()]);
24    ss.setNamedRange('FileDetails',
25                     sh.getRange(2, 1, 2, 3));
26 }
```

Once the function *createFiles()* is executed, it creates two new, useless, and empty files: A spreadsheet and a document. It calls the *getId()* and *getUrl()* methods for both the *Spreadsheet* and *Document* objects and records this information in the active spreadsheet. The methods invoked to get the file IDs and their URLs will be used

in later examples so they are worth noting now. All the other code used to set cell values and formats and to assign a name to a range has been covered in earlier chapters. The sheet populated by this code should look something like the following figure:

	A	B	C	D	E	F	G	H	I	J
1	File Name	File ID	File URL							
2	myspreadsheet	1e0AZigtw0bixu	<a href="https://docs.google.com/a/javascript-spreadsheet-programming.com/spreadsheets/d/1e0AZigtw0bixu/NF88Bn_nOfndb-r8RYCHt5kN84Hg/edit">https://docs.google.com/a/javascript-spreadsheet-programming.com/spreadsheets/d/1e0AZigtw0bixu/NF88Bn_nOfndb-r8RYCHt5kN84Hg/edit</a>							
3	mydocument	1HNCmkShjWU	<a href="https://docs.google.com/a/javascript-spreadsheet-programming.com/open?d=1HNCmkShjWU4RmsW7ee3Cn7u3232ch2SDGskGE24D88">https://docs.google.com/a/javascript-spreadsheet-programming.com/open?d=1HNCmkShjWU4RmsW7ee3Cn7u3232ch2SDGskGE24D88</a>							
4										
5										

Figure 8-2: Recorded details for the newly created files.

The user will see a different file ID and URL but the output should be very similar. The created files are only visible to their owner that being whoever executed the function above. To share these files with others, permissions need to be explicitly granted.

## 8.5.2 Adding Viewers And Editors

Using the information recorded above about the new files, viewers and editors can be added where viewers have read permissions only while editors have read-write permissions. The following function uses the spreadsheet output and the range name from the last code example as input.

### Code Example 8.11

```

1 // Relies on code example 8.10 having been executed.
2 // Gets file ID values from the spreadsheet in figure
3 // 8_2.
4 function addViewer() {
5     var ss = SpreadsheetApp.getActiveSpreadsheet(),
6         rngName = 'FileDetails',
7         inputRng = ss.getRangeByName(rngName),
8         docId = inputRng.getCell(2, 2).getValue(),
9         doc = DriveApp.getFileById(docId),
10        newViewer =

```



```
11         'mick@javascript-spreadsheet-programming.com' ;  
12     doc.addViewer(newViewer);  
13 }
```

Once this function is executed (after being authorised), it adds 'mick@javascript-spreadsheet-programming.com' as a viewer. The viewer is notified by email with a subject containing the text "invitation to view". The sharing effect can be seen in the Google Drives of both the owner and the viewer. For the owner, the "Shared" text flag appears after the file name and the viewer can be seen in the file details shown by the right mouse click described earlier. For the viewer, the file is now listed in his or her Google Drive. Shared files can be listed on their own by selecting "Shared with me" from under the "My Drive" folder on the left hand side of the Google Drive window. This code example confers viewing privileges only so if we want to allow our collaborator to edit the file, we can replace *doc.addViewer* with *doc.addEditor* and our collaborator will be notified by email with a subject containing the text "invitation to edit".

Note: Viewers and editors can be added directly using *Spreadsheet* and *Document* methods but the *DriveApp* mechanism is more general and can be applied to any file types.

### 8.6.3 Removing Viewers And Editors

The code to remove viewers and editors from files is very similar to adding them. Firstly, the file object has to be identified and, as with adding permissions, it is done by retrieving the relevant file ID. To remove the viewer, simply change the line

```
1 doc.addViewer(newViewer);
```

to

```
1 doc.removeViewer(newViewer);
```

in the function *addViewer()*. Similarly for the editor, change the line

```
1 ssToShare.addEditor(newEditor);
2
3 to
4
5 ssToShare.removeEditor(newEditor);
```

## 8.5.4 Folder Permissions

Viewers and editors can be added to folders as easily as to files. The main advantage of using folder-based permissions is that any files added to the folder “inherit” the folder’s permissions without the user having to do anything to the file permissions themselves. In example 8.12 below, we create a new folder and two new files (a sheet and a document), add them to the new folder and then remove them from the root folder (“My Drive”).

### Code Example 8.12

```
1 // Create a new folder and two files:
2 // a sheet and a document.
3 // Add them to the new folder and
4 // remove them from the root folder.
5 function newFilesToShareInNewFolder() {
6     var ss = SpreadsheetApp.getActiveSpreadsheet(),
7         rootFolder = DriveApp.getRootFolder(),
8         newFolder =
9             DriveApp.createFolder('ToShareWithMick'),
10        sh = SpreadsheetApp.create('mysheet'),
11        shId = sh.getId(),
12        shFile =
```

```
13         DriveApp.getFileById(shId),
14         doc = DocumentApp.create('mydocument'),
15         docId = doc.getId(),
16         docFile = DriveApp.getFileById(docId);
17     newFolder.addFile(shFile);
18     newFolder.addFile(docFile);
19     rootFolder.removeFile(shFile);
20     rootFolder.removeFile(docFile);
21 }
```

The function named *newFilesToShareInNewFolder()* creates a new folder, adds the two files to this folder and removes them from the “My Drive” root folder. To re-iterate a point made earlier in the chapter, a file can **belong to more than one folder**. Also, when files are added to folders (the *Folder addFile()* method), they **are not copied!** The association is more like a reference where the folders store references to files and not copies of them.

Once the folder has been created and the target files have been added to it, it is quite simple to add viewers and editors. The following function retrieves the *Folder* object using the ID from the URL and then adds a viewer to it.

### Code Example 8.13

```
1 // Add a viewer to a folder.
2 // Replace ID with one from a folder URL and
3 // the email address in the "addViewer()* method
4 // call.
5 function shareFolder() {
6     var folderId = 'ID From Url',
7         folder = DriveApp.getFolderById(folderId);
8     folder.addViewer('emailAddress');
9 }
```

To execute the code in example 8.13, replace the *folderId* with your

own (for example, the URL looks something like this: <https://drive.google.com/drive/folders/0B2dsdq7IKB9yem03VWZGdTNvRFE> in this example) and replace the string “emailAddress” with the address of whom you want to allow view the folder and its contents.

In Google Drive, folder names can be duplicated in the same way as file names so I find it better to work with IDs because these are guaranteed to be unique.

Folder permissions are very convenient. Files can be grouped conveniently into folders and when the permissions are set at the folder level, the files get those permissions. The same group of files within a folder could have one associated list of viewers and another associated list of editors. An additional convenience is that the *File* and *Folder* methods *addViewer()* and *addEditor()* have plural equivalents that take arrays of email addresses so that you can add multiple viewers/editors in one go.

## 8.6 Practical Examples

Having covered the basics of folder and file creation, deletion and permissions, we can consolidate this knowledge with some practical examples

### 8.6.1 Get All Sub-Folders

Example 8.1 showed how to use the *FolderIterator* object to print all folder names to the logger. Just printing a list is not so useful. Something more useful is a function that can return an array of *Folder* objects for a given folder, that is, we want an array of all the sub-folders for a given parent folder. Since folders can be nested to an arbitrary depth in a folder/sub-folder hierarchy, we want a function that can find and return all sub-folders. The programming

technique of **recursion**<sup>64</sup> is ideal for processing nested structures like file shares. The function `getAllSubFolders()` in example 8.14 is a recursive function as will be explained below.

#### Code Example 8.14

```
1  // Given a Folder object and an array,
2  // recursively search for all folders
3  // and return an array of Folder objects.
4  function getAllSubFolders(parent, folders) {
5      var folderIt = parent.getFolders(),
6          subFolder;
7      while (folderIt.hasNext()) {
8          subFolder = folderIt.next();
9          getAllSubFolders(subFolder, folders);
10         folders.push(subFolder);
11     }
12     return folders;
13 }
14 // Execute function "getAllSubFolders()" passing
15 // it the "root" folder.
16 function run_getAllSubFolders() {
17     var root = DriveApp.getRootFolder(),
18         folders = [],
19         folders = getAllSubFolders(root, folders);
20     folders.forEach(function(folder) {
21         Logger.log(folder.getName());
22     });
23 }
```

Function `getAllSubFolders()` returns an array of *Folder* objects where each array element is a sub-folder of the *parent* folder argument. This function can be combined with other functions in useful ways

---

<sup>64</sup>[https://en.wikipedia.org/wiki/Recursion\\_%28computer\\_science%29](https://en.wikipedia.org/wiki/Recursion_%28computer_science%29)

as I will show in later examples. Its most interesting and novel aspect in terms of this book is the line `getAllSubFolders(subFolder, folders)`; This is the recursive call where the function calls itself. Without this recursive call, we would not retrieve the “grandchild” folders of the *parent* argument (sub-folders of sub-folders). Also, if we make the second *folders* array argument a variable within the function body, the code will execute but it will not act recursively and we will only return the immediate child folders of the *parent* argument. Recursion is an interesting topic in its own right but many academic examples, such as Fibonacci sequences, are not very practical.

## 8.6.2 Checking If A Folder Name already Exists

Before creating a new folder, we might wish to know if a folder with that name already exists in target location thereby preventing us from creating duplicates. An application could then use it before creating a folder and throw an exception if an attempt is made to create a folder with a duplicate name. Code example 8.15 provides a function that does just that.

### Code Example 8.15

```

1 // Given a parent Folder object and a folder name,
2 // call the function "getAllSubFolders()" to return
3 // an array of Folder objects. The array "map" method
4 // is used to return the folder names as an array.
5 // This array is checked to determine if the folder name
6 // is a member.
7 function folderNameExists(parentFolder,
8                             subFolderName) {
9     var folderNames = [],
10         folders = getAllSubFolders(parentFolder, []);
11     folderNames = folders.map(
12         function (folder) {
```

```
13         return folder.getName();
14     });
15     if (folderNames
16         .indexOf(subFolderName) > -1) {
17         return true;
18     } else {
19         return false;
20     }
21 }
22 // Code to test "folderNameExists()"
23 // Change the value of folderName to existing
24 // and non-existent folder names to see
25 // how the tested function operates!
26 function test_folderNameExists() {
27     var folder = DriveApp.getRootFolder(),
28         folderName = 'JavaScript';
29     if (folderNameExists(folder, folderName)) {
30         Logger.log("yup");
31     } else {
32         Logger.log("Nope");
33     }
34 }
```

The function *folderNameExists()* can be used in GAS applications to test for folder name existence before new folders are created to avoid inadvertently creating a folder with a pre-existing name. Note, that it calls the function *getAllSubFolders()* that was defined in code example 8.14. It uses the *Array map()* function to traverse the array of *Folder* objects to extract the folder names. Once we have the array of names, we can use another array method, *indexOf()* in this example, to determine if the folder name already exists as an element of the array.

The function return value can be used by the calling code to determine the appropriate course of action. For example, an exception

could be thrown or a warning issued. Note also that the code does a **case-sensitive** search for the given sub-folder name but it could be easily amended to perform a case-insensitive search. A similar function could be written to check if given file names already exist in a given folder.

### 8.6.3 Retrieve The Names And File IDs Of All Files Older Than Given Date

Filtering files by age is a common task in conventional file systems. For example, there may be processes in place to archive or remove files older than a certain date. The following GAS code provides a function to return an array of all *File* objects in the user's Google Drive older than the given cut-off date and a function to write a selection of file properties for the filtered array of *File* objects to a newly created sheet.

#### Code Example 8.16

```
1 // Filter all files based on a given cut-off date
2 // and return an array of files older than the
3 // cut-off date.
4 function getFilesOlderThan(cutoffDate) {
5     var fileIt = DriveApp.getFiles(),
6         filesOlderThan = [],
7         file;
8     while(fileIt.hasNext()) {
9         file = fileIt.next();
10        Logger.log(file);
11        if(file.getDateCreated() < cutoffDate) {
12            filesOlderThan.push(file);
13        }
14    }
15    return filesOlderThan;
16 }
```



```
17
18 // Write some file details for files older than
19 // a specified date to a new sheet.
20 function test_getFilesOlderThan() {
21     var ss = SpreadsheetApp.getActiveSpreadsheet(),
22         shName = 'OldFiles',
23         sh = ss.insertSheet(),
24         // 'July 28, 2014', months are 0-11 in JS
25         testDate = new Date(2014, 8, 23);
26         oldFiles = getFilesOlderThan(testDate);
27     sh.setName(shName);
28     oldFiles.forEach(
29         function (file) {
30             sh.appendRow([file.getName(),
31                           file.getSize(),
32                           file.getDateCreated(),
33                           file.getId()]);
34         });
35 }
```

When you execute function *test\_getFilesOlderThan()*, a new sheet called “oldFiles” should be added to the active spreadsheet that will write selected details for all files older than the *testDate* value. All of the code techniques used in example 8.16 have been described earlier so they should be reasonably easy to understand. In addition, you should be able to adapt the examples to suit your own needs by, for example, using other *File* methods. One *File* method that may return initially puzzling results is *getSize()*. For Google documents and spreadsheets this method always returns zero! This does not mean that the files are empty, but rather reflects the fact that they are not regular files in the way that, for example, PDFs or Excel files stored in Google Drive are. The size returned for these latter file types denotes how much of the Google Drive storage quota that they use.

## 8.6.4 File Age In Days

The file age in days is easy to determine by using JavaScript's *Date* type. When one date is subtracted from another, the result is given in milliseconds and this can be converted to days. Here is the example code:

### Code Example 8.17

```
1 // Given a File object, return number
2 // of days since its creation.
3 function getFileAgeInDays(file) {
4   var today = new Date(),
5       createdAt = file.getDateCreated(),
6       msecPerDay = 1000 * 60 * 60 * 24,
7       fileAgeInDays =
8         (today - createdAt)/msecPerDay;
9   return Math.round(
10    fileAgeInDays).toFixed(0);
11 }
12
13 // Get a test file ID from the URL
14 // and assign it to variable fileId
15 // Run and check the log
16 function test_getFileAgeInDays() {
17   var fileId = '0B2dsdq7IKB9ydXNBZg2czJpYTg',
18       file =
19         DriveApp.getFileById(fileId),
20       fileAgeInDays =
21         getFileAgeInDays(file);
22   Logger.log('File ' +
23             file.getName() +
24             ' is ' +
25             fileAgeInDays +
26             ' days old.');
```

To run this code, you will need to assign your own file ID to the variable *fileId* above. This can be extracted from the file URL as described earlier.

### 8.6.5 Identify Empty Folders

A folder is considered empty if it has no sub-folders and no files associated with it. They can easily be accumulated in a heavily-used Google Drive so it is occasionally useful to identify and remove them. I was surprised when I ran this code on my own Google Drive to learn how many of my folders were actually empty. To check your own Google Drive for empty files, execute function *processEmptyFolders()* in code example 8.18 and check the log output for details on empty folders.

#### Code Example 8.18

```
1  // Return an array of File objects
2  // for a given Folder argument.
3  function getFilesForFolder(folder) {
4      var fileIt = folder.getFiles(),
5          file,
6          files = [];
7      while(fileIt.hasNext()) {
8          file = fileIt.next();
9          files.push(file);
10     }
11     return files;
12 }
13 // Return an array of empty folder objects
14 // in the user's Google drive.
15 // An empty folder is one with no associated
16 // files or sub-folders.
17 // NB: Requires function "getAllSubFolders()"
18 // from Code Example 8.14.
```

```
19 function getEmptyFolders() {
20     var folderIt = DriveApp.getFolders(),
21         folder,
22         emptyFolders = [],
23         files = [],
24         folders = [];
25     while(folderIt.hasNext()){
26         folder = folderIt.next();
27         files = getFilesForFolder(folder);
28         folders = getAllSubFolders(folder, []);
29         if(files.length === 0 && folders.length === 0) {
30             emptyFolders.push(folder);
31         }
32     }
33     return emptyFolders;
34 }
35 // Write the IDs and names of
36 // all empty folders to the
37 // log.
38 function processEmptyFolders() {
39     var emptyFolders =
40         getEmptyFolders();
41     emptyFolders.forEach(
42         function (folder) {
43             Logger.log(folder.getName() +
44                 ': ' +
45                 folder.getId());
46         });
47 }
```

I have added a small utility function called *getFilesForFolder()* that returns an array of *File* objects for the given *Folder* argument. Likewise, we can check folders for sub-folders using the function *getAllSubFolders()* that was defined in code example 8.14. To determine folder “emptiness”, each *Folder* object is checked to see if both

*getFilesForFolder()* and *getAllSubFolders()* returned empty arrays.

The array of empty *Folder* objects could be subjected to more sophisticated processing than that shown above. Empty folders could be removed, for example. An obvious limitation of this example is how it deals with folders that have only empty sub-folders. Once those empty sub-folders are removed, their parent folders would also become empty but the code would have to be re-run to identify them. However, despite these limitations, this function is still quite useful and enhancing it should not prove very difficult.

### 8.6.5 Identify Files With Duplicate Names

This task involves quite a lot of Google Apps Script code so it is presented and discussed in pieces to make it more accessible.

Function *getFileNameIdMap()* loops through all the files in the current Google Drive and builds a JavaScript object with the file names as keys mapping to an array of file IDs. Once all the files have been processed, it returns the object. You can try it out by executing function *getFileNameIdMap()*.

#### Code Example 8.19 - A

```
1 // Return object mapping each
2 // file name to an array of file IDs.
3 // If file names are not duplicated,
4 // the arrays they map to will
5 // have a single element (file ID).
6 function getFileNameIdMap() {
7     var fileIt = DriveApp.getFiles(),
8         fileNameIdMap = {},
9         file,
10        fileName,
11        fileId;
12    while(fileIt.hasNext()) {
```

```
13     file = fileIt.next();
14     fileName = file.getName();
15     fileId = file.getId();
16     if (fileName in fileNameIdMap) {
17         fileNameIdMap[fileName]
18             .push(fileId);
19     } else {
20         fileNameIdMap[fileName] = [];
21         fileNameIdMap[fileName]
22             .push(fileId);
23     }
24 }
25 return fileNameIdMap;
26 }
27 // Execute to test "getFileNameIdMap()".
28 function run_getFileNameIdMap() {
29     var fileNameIdMap = getFileNameIdMap();
30     Logger.log(fileNameIdMap);
31 }
```

This function is useful in its own right since any calling code can use it to determine if a file name exists or if the file name is associated with more than one file ID. It also demonstrates the flexibility of JavaScript's objects and how easily they can be used to store complex data structures. It is worth taking a little time to study the *if* statement within the *while* loop above. It tests if the file name already exists as a key in the object, if it does, it pushes the corresponding file ID onto the array it maps to. If the file name does not exist as a key, the key is created and mapped to a single-element array containing the file ID. If the same file name is found in a subsequent iteration, the array that it maps to will have the new file ID for this file name added to it.

Most of the file names will map to just a single file ID and their mapped arrays in will therefore be single element. In order to

filter out those file names that map to multiple IDs, the object mapping the names to arrays of IDs can be processed in function *getDuplicateFileNamesIds()* in code example 8.19 - B below.

### Code Example 8.19 - B

```
1 // Code Example 8.19 - B
2 // Return an array of file IDs for files
3 // with duplicate names.
4 // Loops over the object returned by
5 // getFileNameIdMap() and returns
6 // only those file IDs for duplicate
7 // file names.
8 function getDuplicateFileNamesIds() {
9     var fileNameIdMap = getFileNameIdMap(),
10     fileName,
11     duplicateFileNamesIds = [];
12     for (fileName in fileNameIdMap) {
13         if (fileNameIdMap[fileName].length > 1) {
14             duplicateFileNamesIds =
15                 duplicateFileNamesIds
16                 .concat(fileNameIdMap[fileName]);
17         }
18     }
19     return duplicateFileNamesIds;
20 }
21 // Run function getDuplicateFileNamesIds().
22 function run_getDuplicateFileNamesIds() {
23     var duplicateFileNamesIds =
24         getDuplicateFileNamesIds();
25     Logger.log(duplicateFileNamesIds);
26 }
```

Function *getDuplicateFileNamesIds()* loops over the keys of the object returned by *getFileNameIdMap()* and, if the key maps to an

array with more than one element, it takes the array values and adds them to a new array. It uses the array `concat()` method in order to generate a one-dimensional array. This array of file IDs is returned after all the keys have been checked.

An array of file IDs for those files with duplicated file names is all that is required to identify the files and take some action. The function `addDuplicateFilesToFolder()` defined in code Example 8.18 - C below creates a folder for grouping the duplicated files and uses the array of file IDs to create *File* objects. Each of these *File* objects is then added to the newly created folder.

### Code Example 8.19 - C

```
1 // Add files with duplicated names to
2 // a newly created folder.
3 function addDuplicateFilesToFolder() {
4     var duplicateFileNameIds =
5         getDuplicateFileNameIds(),
6         folderName = 'DUPLICATE FILES',
7         folder =
8             DriveApp.createFolder(folderName);
9     duplicateFileNameIds.forEach(
10        function (fileId) {
11            var file =
12                DriveApp.getFileById(fileId);
13            folder.addFile(file);
14        });
15 }
```

For the purposes of this example, the files with duplicated names were simply added to a folder that was created to flag them. Other actions could also have been taken such as deleting the older file of the duplicate pair or the list could have been written to a spreadsheet.



## 8.7 Summary

- Google Drive file and folder contents can be interrogated and manipulated using the *DriveApp* object.
- The *DriveApp* object has methods that return iterator objects. The returned iterators can be traversed in a *while* loop using their *next()* method to obtain *File* and *Folder* objects.
- *File* and *Folder* objects have a large number of methods to perform such tasks as folder/file deletion, user permission changes, and adding of files to folders and folders to other folders.
- *File* and *Folder* objects are uniquely defined by their IDs and not by their names and locations. Their names are not required to be unique within their parent folders.
- The top-level *Folder* in Google Drive is called “My Drive”.
- A file can belong to any number of folders and its parent folders can be retrieved using the *File getParents()* method.
- Permission to view or view and edit files can be granted to other Gmail account users for both files and folders.
- Files are assigned the permissions of their parent folders. This makes folders suitable for grouping files and their permissions.
- GAS can be used to automate Google Drive tasks and to add new functionality.
- This chapter provides a number of practical GAS examples that can be adapted and extended to enhance functionality.

# Chapter 9: Email and Calendars

## 9.1 Introduction

Apart from online searching, Gmail is what most people associate with Google. Of course, a Gmail account comes with the calendar and calendars are shared and Gmail is used to send calendar notifications and reminders. These user interactions are central to the online collaborative nature of Google products. GAS can be used to send emails, read emails, save email attachments to Google Drive, as well as read and update calendars. Scripts written in Google Spreadsheets can use spreadsheets as a convenient backend store for lists of email addresses and contacts lists. This of course makes GAS a potential spamming tool so Google have put in place measures to prevent this in the form of email quotas. Email and calendars are the subject of this chapter, specifically, the three GAS objects *MailApp*, *GMailApp* and *CalendarApp*.

**Some general cautionary words on sending emails with URLs or attachments.** Users have rightly become suspicious of unsolicited emails, especially if they contain links or attachments. Even when they trust the sender, they may still be reluctant to open such mails and their contents for fear that the sender's account may have been compromised. The types of scripts shown here should not then be abused as spamming or scamming tools. Remember the Google motto: "Don't be evil"!

All code for this chapter is available on GitHub at [this location](#)<sup>65</sup>. Create a new sheet and paste all this code into its script editor and

---

<sup>65</sup>[https://github.com/Rotifer/GoogleSpreadsheetProgramming\\_2015/blob/master/ch09.gs](https://github.com/Rotifer/GoogleSpreadsheetProgramming_2015/blob/master/ch09.gs)

save it. You can then run the examples as they are discussed below.

## 9.2 Sending An Email Using *MailApp*

The *MailApp* object is very easy to use. To demonstrate it, we will first create a function that we can use to return an array of email addresses that we can feed to *MailApp*.

### Code Example 9.1

```
1 // Code Example 9.1
2 // Use a named range (name = "EmailContacts")
3 // to return an array of email addresses.
4 // All the code techniques used here have been
5 // covered in earlier examples.
6 // To use: Create a list of email addresses
7 // in a single column, give that list the range
8 // name "EmailContacts". Execute function
9 // "run_getEmailList()" to ensure it is working.
10 function getEmailList() {
11     var ss =
12         SpreadsheetApp.getActiveSpreadsheet(),
13         rngName = 'EmailContacts',
14         emailRng = ss.getRangeByName(rngName),
15         rngValues = emailRng.getValues(),
16         emails = [];
17     emails = rngValues.map(
18         function (row) {
19             return row[0];
20         });
21     return emails;
22 }
23 // Check that function "getEmailList()"
24 // is working.
```

```

25 function run_getEmailList() {
26     Browser.msgBox(getEmailList().join(','));
27 }

```

We can now use the return value (array of email addresses) of function *getEmailList()* for input into *MailApp*.

And now for the main business of this example. Equipped with a list of email addresses in a spreadsheet that can be retrieved as an array and a function to add each of these email accounts as editors to a folder, the function that follows creates the new folder, adds the editors, and emails each of the editors to inform them of the action.

### Code Example 9.2

```

1  // Code Example 9.2
2  // Need to authorize!
3  // Get an array of email addresses using function
4  // "getEmailList()" in code example 9.1 and send
5  // a test email to them.
6  function sendEmail (){
7      var emailList = getEmailList(),
8          subject = 'Testing MailApp',
9          body = 'Hi all,\n' +
10             'Just checking that our email distribution list\
11 is working!\n' +
12             'Cheers\n' +
13             'Mick';
14     MailApp.sendEmail(emailList.join(','),
15                       subject,
16                       body);
17 }
18 // Display remaining daily quota.
19 function showDailyQuota() {
20     Browser.msgBox(MailApp.getRemainingDailyQuota());
21 }

```

This function introduces the very useful *MailApp* object that can be used in Google Apps Script to send emails and email attachments. Its *sendEmail()* method has multiple overloads and the overload executed above is one of the easiest to use. It takes only three arguments: (1) A comma-separated string of email address (this explains the call to the array *join()* method), (2) A text describing the subject of the email, and (3) The email body itself.

For many applications the *MailApp* object may be all that is needed. In addition to the various overloads for the *sendEmail()* method, it has only one other method named *getRemainingDailyQuota()* as demonstrated in the *showDailyQuota()* function. It returns an integer value representing the remaining number of emails that the user can send. For the account used here, it starts at 100. Each recipient of an email counts as one for this quota so sending one email to 100 recipients would consume a user's entire daily quota.

### 9.3 Sending An Email With An Attachment Using *MailApp*

Sharing documents and folders reduces the requirement to email attachments to collaborators. However, there are occasions when attachments are unavoidable. Gmail accounts may not be available to certain individuals because some companies block Google Documents entirely as a potential security risk. In such situations, it may be necessary to attach the file to the email. Sending a Google document such as a spreadsheet as an attachment is almost always pointless because, without a Google account, the user will not be able to open it anyway. Currently, it is not possible to attach Microsoft Word or Excel files using either *MailApp* or *GMailApp*. A few possible alternative strategies are discussed below.

In order to attach a file, its Multipurpose Internet Mail Extensions (MIME) type must be specified and *MailApp* currently accepts a limited set of MIME types: These are Portable Document Format

(PDF) and some image types. PDFs are fine for reading, especially when editing is neither needed nor allowed. However, for spreadsheet applications, they are generally not much use. The ability to use Excel as a MIME type has been requested of Google so it may appear in the future.

In order to demonstrate the general technique of attaching and sending a file, here is an example that creates a Google word processing document and then retrieves it as a PDF before sending it as an attachment.

### Code Example 9.3

```
1 // Uses function "getEmailList()* from
2 // code example 9.1.
3 // Creates a Google document and converts it to
4 // a PDF that it then sends to the email list
5 // as an attachment.
6 function sendAttachment() {
7     var doc
8         = DocumentApp.create('ToSendAsAttachment'),
9         emailList = getEmailList(),
10        file,
11        pdf,
12        pdfName = 'Test.pdf',
13        fileId = doc.getId(),
14        subject = 'Test attachment',
15        body = 'See attached PDF',
16        attachment,
17        paraText =
18        'This is text that will be written\n' +
19        'to a document that will then be saved\n' +
20        'as a PDF and sent as an attachment. ';
21    doc.appendParagraph(paraText);
22    doc.saveAndClose();
23    file = DriveApp.getFileById(fileId);
```

```
24 pdf = file.getAs('application/pdf').getBytes();
25 attachment = {fileName: pdfName,
26               content:pdf,
27               mimeType:'application/pdf'};
28 MailApp.sendEmail(emailList.join(','),
29                  subject,
30                  body,
31                  {attachments:[attachment]});
32 }
```

Executing the above function will send the newly created PDF file as an attachment to the email addresses returned by the function *getEmailList()* that was defined in code example 9.1. This function uses *DocumentApp* methods to create a document that is then converted into a PDF by the *DriveApp File* method *getAs()*.

That is all that will be covered regarding *MailApp*. The *GmailApp* object offers much more functionality in addition to just sending emails and attachments. It does not however, address the shortcomings of *MailApp* when it comes to sending attachments. That limitation may be addressed in the near future.

## 9.4 GmailApp

Google Apps Script email interaction is not just confined to sending emails and attachments. The *GmailApp* object can do a lot more than this. In fact it is quite complex so this section will focus on just four subjects: threads, messages, labels, and attachments. An email thread is composed of one or more messages and each message belongs to a thread and can have zero or more attachments.

## 9.4.1 Email Threads

A thread in the email sense (not to be confused with a thread of execution) is composed of one or more email messages. When someone replies to a message for the first time, the thread will contain two messages.

An array of *GmailThread* objects is returned by the *GmailApp* *getInboxThreads()* method. *GmailThread* objects, in common with other Google Apps Script objects, have a large number of methods. A few of the more important ones are demonstrated by the function given below. This function takes the user's mail box and generates an array of objects that it returns. Each of the array objects stores some important information about that thread.

### Code Example 9.4

```
1 // Return a list of objects that contain
2 // selected details of the user's email
3 // threads.
4 // This will be slow for a large mail box
5 // so there is an optional argument to limit
6 // the returned array to a certain number.
7 // The most recent threads are returned first.
8 function getThreadSummaries(threadCount) {
9     var threads =
10         GmailApp.getInboxThreads(),
11         threadSummaries = [],
12         threadSummary = {},
13         recCount = 0;
14     threads.forEach(
15         function (thread) {
16             recCount += 1;
17             if (recCount > threadCount) {
18                 return;
19             }
20         }
21     );
22 }
```



```
20     threadSummary = {};  
21     threadSummary['MessageCount'] =  
22         thread.getMessageCount();  
23     threadSummary['Subject'] =  
24         thread.getFirstMessageSubject();  
25     threadSummary['ThreadId'] =  
26         thread.getId();  
27     threadSummary['LastUpdate'] =  
28         thread.getLastMessageDate();  
29     threadSummary['URL'] =  
30         thread.getPermalink();  
31     threadSummaries  
32         .push(threadSummary);  
33     });  
34     return threadSummaries;  
35 }  
36 // Run "getThreadSummaries()" for 50 threads.  
37 // Check the log after running.  
38 function run_getThreadSummaries() {  
39     Logger.log(getThreadSummaries(50));  
40 }
```

Looping over the entire collection of threads for a heavily used Gmail account can take some time so this function allows an optional integer argument and returns summaries for just that number of the most recent threads. If the optional argument is omitted or a non-numeric value is passed in, the function will return an array of objects for the entire Gmail account!

The function `getThreadSummaries()` simply writes the output to the log. It can more usefully be written to a sheet as shown in code example 9.5. This example inserts a new sheet named “EmailThreadSummary” into the active spreadsheet, writes a column header row to this sheet, makes the header column font bold, and then loops over the array of objects and writes the object data

to the sheet.

### Code Example 9.5

```
1 // Write some thread details returned by
2 // function "getThreadSummaries()" to
3 // a newly inserted sheet named
4 // "EmailThreadSummary"
5 // Make sure the sheet is deleted before
6 // re-running or an error will be thrown
7 // when it attempts to insert a new sheet
8 // with the same name.
9 function writeThreadSummary() {
10     var threadCount = 10,
11         threadSummaries =
12         getThreadSummaries(threadCount),
13         ss =
14         SpreadsheetApp.getActiveSpreadsheet(),
15         sh = ss.insertSheet(),
16         headerRow;
17     sh.setName('EmailThreadSummary');
18     sh.appendRow(['Subject',
19                 'MessageCount',
20                 'LastUpdate',
21                 'ThreadId',
22                 'URL']);
23     headerRow = sh.getRange(1,1,1,5);
24     headerRow.setFontWeight('bold');
25     threadSummaries.forEach(
26     function (threadSummary) {
27         sh.appendRow([threadSummary['Subject'],
28                     threadSummary['MessageCount'],
29                     threadSummary['LastUpdate'],
30                     threadSummary['ThreadId'],
31                     threadSummary['URL']]);
```

```
32     });  
33 }
```

All the techniques used in example 9.5 should be quite familiar by now. When this example is executed, a new sheet should appear in the host spreadsheet that has five columns that contain email thread-specific information. For demonstration purposes, the thread count in this example has been set to a value of 10. The function that returns the array of objects that contain selected information about the threads could be adapted to do custom filtering such as only returning information for threads between certain dates.

## 9.4.2 Email Messages

No mention was made above in the email thread discussion of who the email senders were or what the email messages contained. That information can be extracted from the *GmailMessage* object and the *GmailMessage* objects themselves are contained within the *GmailThread* object.

The example code that follows shows how to process the 10 email threads written to the sheet named “EmailThreadSummary” in the previous code example. There are three functions and the code in the last one that actually writes the output is quite complex so the explanation will be more extensive than usual.

The first function in the code that follows takes a thread ID as an argument and uses this ID to retrieve all associated *GmailMessage* objects. It operates in a similar fashion to the function *getThreadSummaries()* in that it builds an array of objects where the object properties are values extracted using *GmailMessage* methods.

**Code Example 9.6 - A**

```
1 // Create an array of objects containing
2 // information extracted from each GmailMessage
3 // object for the given a thread ID.
4 // Each object property is populated by
5 // the return value of a GmailMessage
6 // method.
7 // Return 'undefined' if there are no
8 // messages.
9 function getMsgsForThread(threadId) {
10   var thread =
11     GmailApp.getThreadById(threadId),
12     messages,
13     msgSummaries = [],
14     messageSummary = {};
15   if (!thread) {
16     return;
17   }
18   messages = thread.getMessages();
19   messages.forEach(
20     function (message) {
21       messageSummary = {};
22       messageSummary['From'] =
23         message.getFrom();
24       messageSummary['Cced'] =
25         message.getCc();
26       messageSummary['Date'] =
27         message.getDate();
28       messageSummary['Body'] =
29         message.getPlainBody();
30       messageSummary['MsgId'] =
31         message.getId();
32       msgSummaries.push(messageSummary);
33     });
34   return msgSummaries;
35 }
```

Other values could be extracted from the *GmailMessage* objects but the ones returned above will suffice for this example. In the next function each of the objects returned in the array is passed to another function that then writes the property values to a Google *document*. A Google document was chosen instead of a Google Sheet because the long text in the body of the email message is not much use in a spreadsheet cell. Here is the function that implements this functionality:

### Code example 9.6 - B

```
1 // Given a message summary object as returned
2 // in an array by "getMsgsForThread" and
3 // a Google Document ID, open the document,
4 // extract the object values and write them
5 // to the document.
6 // Argument checks could/should be added to
7 // ensure that the first argument is an
8 // object and that the second is a valid
9 // ID for a document.
10 function writeMsgBodyToDoc(msgSummsForThread,
11                          docId) {
12   var doc = DocumentApp.openById(docId),
13       header,
14       from =
15         msgSummsForThread['From'],
16       msgDate =
17         msgSummsForThread['Date'],
18       msgBody =
19         msgSummsForThread['Body'],
20       msgId =
21         msgSummsForThread['MsgId'],
22       docBody = doc.getBody(),
23       paraTitle;
24   docBody.appendParagraph('From: ' +
```

```

25         from +
26         '\rDate: ' +
27         msgDate +
28         '\rMessage ID: ' +
29         msgId);
30     docBody.appendParagraph(msgBody);
31     docBody
32     .appendParagraph(' ***** ');
33     doc.saveAndClose();
34 }

```

In order to actually see the document with the email message information written to it, a third function is required. This function may look a little intimidating but it can be summarised as follows:

- Get a reference to the *Sheet* object named “EmailThreadSummary” created earlier in code example 9.5.
- Create a document entitled “ThreadMessages” (line *DocumentApp.create(docName)*).
- Define the range containing the *GmailThread* IDs using the *Sheet getRange()* method (column D).
- The outer *forEach* array method extracts the *GmailThread* ID from the nested array returned by the *Range getValues()* method and passes this ID as an argument to the function *getMsgsForThread()*. This function returns another array of objects containing summary information for each message in that thread.
- The inner *forEach()* array method processes each message summary object by passing it and the ID for the newly created document to function *writeMsgBodyToDoc()*.

### Code Example 9.6 - C

```
1 // Use the Thread IDs generated earlier
2 // to extract the message for each thread
3 // Create a new Google Document
4 // Write summary message data to the Google
5 // Document.
6 // Contains a nested forEach structure,
7 // see explanation in text.
8 function writeMsgsForThreads() {
9   var ss =
10     SpreadsheetApp.getActiveSpreadsheet(),
11     shName = 'EmailThreadSummary',
12     sh = ss.getSheetByName(shName),
13     docName = 'ThreadMessages',
14     doc =
15     DocumentApp.create(docName),
16     docId = doc.getId(),
17     threadIdCount =
18     sh.getLastRow() - 1,
19     rngThreadIds =
20     sh.getRange(2,
21               4,
22               threadIdCount,
23               1),
24     threadIds =
25     rngThreadIds.getValues();
26   threadIds.forEach(
27     function(row) {
28       var threadId
29       = row[0],
30       msgsForThread =
31       getMsgsForThread(threadId);
32       msgsForThread.forEach(
33         function(msg) {
34           writeMsgBodyToDoc(
35             msg,
```

```
36         docId);
37     });
38 });
39
40 }
```

Executing function *writeMsgsForThreads()* in code example 9.6 part C creates a new Google Document in your Drive with various email details. This is a complex example but it does showcase how you can use GAS to build applications that can manipulate a variety of Google products; in this example, we used *GmailApp*, *DocumentApp* and *SpreadsheetApp* objects to create the final output.

### 9.4.3 Email Labels

Labels can be added to email threads either manually from the Gmail mail view or programmatically using GAS. Labels are a type of flag that can be used to categorise emails based on criteria such as the original message sender, the subject title, and so on. Once you have added a label (manually or programmatically), you will see the label on the left-hand side of your Gmail mail view. You can then filter your emails by selecting the label. Code example 9.7 demonstrates how to add a label with a given text based on the email subject.

#### Code Example 9.7



```
1 // Add a label text as given in the first argument
2 // to all email threads where
3 // the first message in the thread has
4 // subject text that matches the first
5 // argument in its subject section.
6 // To do this:
7 // 1: Create a label object using the
8 //     GmailApp createLabel() method.
9 // 2: Retrieve all the inbox threads.
10 // 3: Filter the resulting array of
11 //     threads based on the specified
12 //     subject text.
13 // 4: Add the label to each of the
14 //     threads in the filtered array
15 // 5: Call the thread refresh method
16 //     to display the label.
17 function labelAdd(subjectText, labelText) {
18     var label =
19         GmailApp.createLabel(labelText),
20         threadsAll =
21             GmailApp.getInboxThreads(),
22         threadsToLabel;
23     threadsToLabel = threadsAll.filter(
24         function (thread) {
25             return (thread.getFirstMessageSubject()
26                 ===
27                 subjectText);
28         });
29     threadsToLabel.forEach(
30         function (thread) {
31             thread.addLabel(label);
32             thread.refresh();
33         });
34 }
```

The code comments describe the series of actions effected by the function `addLabel()`. Labels are objects of type `GmailLabel` and so have their own methods. The code above used the `GmailThread` `addLabel()` method taking a `GmailLabel` object as an argument. Alternatively, the `GmailLabel` method `addToThread()` could have been used taking a `GmailThread` object as its argument. Once a thread has been added, it can be removed by calling the `GmailThread` method `removeLabel()` passing it a `GmailLabel` object as an argument. As with adding labels to threads, labels also possess methods to remove themselves from `GmailThread` objects.

In addition to appearing in the left side of the Gmail window, each of the email threads also appears with the label

Beware that looping over and altering email threads for an entire Gmail inbox can be very slow. The target inbox in this example is quite small but the code still took an appreciable amount of time to run.



### **Slow Loops!**

Looping over all email threads in a big inbox is slow!

#### **9.4.4 Saving Email Attachments To Google Drive**

Saving email attachments to Google Drive is a routine task that we can automate in GAS. In order to demonstrate this, I have sent myself an email with a an attached Word Document to which I then replied. This action results in a thread containing two messages. To simplify matters, I have extracted the thread ID from the URL. This is another recurring GAS theme where IDs such as those for files and folders etc can be determined programmatically or taken from the URL. In this example, the URL in question was:

<https://mail.google.com/mail/ca/#inbox/150195c16476de2a>

The ID portion of this URL is “150195c16476de2a”.

The following function is used to save the attached Word document to the user’s Google Drive:

### Code Example 9.8

```
1 // Using a hard-code thread ID taken from
2 // the email URL, extract the first message
3 // from the thread and extract the first
4 // attachment from that message.
5 // Copy the attachment into a Blob.
6 // Use the DriveApp object to create a File
7 // object from this blob.
8 function putAttachmentInGoogleDrive() {
9     var threadId = '150195c16476de2a',
10         thread =
11         GmailApp.getThreadById(threadId),
12         firstMsg =
13         thread.getMessages()[0],
14         firstAttachment =
15         firstMsg.getAttachments()[0],
16         blob =
17         firstAttachment.copyBlob();
18     DriveApp.createFile(blob);
19     Logger.log('File named ' +
20         firstAttachment.getName() +
21         ' has been saved to Google Drive');
22 }
```

This code is for illustrative purposes only. It uses a thread ID taken from the email URL and then uses prior knowledge on my part to identify the first email in the thread and the first and only attachment in this email message. A more realistic example would require looping over the messages and extracting the attachments. The *DriveApp* object was discussed at length in the last chapter

where usage of folders was also discussed. This knowledge could be applied in this example to save the attachments to specific folders. The most important point to note is that the *GmailMessage* object returns an array of *GmailAttachment* objects. This attachment was copied as a blob (binary large object) and the *DriveApp createFile()* method was then able to take the blob and render it as a Word file. To see this in action, send yourself an email with an attached Excel or Word file. Get the thread ID from the received email URL and substitute it for the one above. Run the code and then check the log for the feedback message and check your Google Drive for the file. If the file name does not display, search for the it in the Google Drive Search box! Check also that the Word or Excel file has survived the “email-detach-drive” process intact.

## 9.5 Calendars

Unsurprisingly, GAS can also be used to manipulate calendars which the user owns or to which he/she has access. Each Gmail account comes with a default calendar and Gmail accounts can subscribe to other calendars. For example, in work, individuals may have access to a department calendar where the can add appointments and events as well as view the status of fellow workers.

The *CalendarApp* object is the key player when it comes to programming Gmail calendars. It is used to create calendar event objects type *CalendarEvent*. The events discussed here are not to be confused with “click” or “mousedown” types of events associated with event-driven programming. Calendar events are basically appointments. All the event options that can be set manually from the calendar interface can also be set programmatically. *CalendarEvent* objects possess a rich set of methods to implement all this functionality; there were 56 of them at the time of writing. The examples given here can only begin to address what is available but

they should provide a basis for further exploration. They should also showcase how spreadsheets can be very useful for interacting with a calendar for providing input and as a destination for output.

### 9.5.1 summary Calendar Information

The function below prints some information to the log about the calendars available to the user. It creates an array of *Calendar* objects and then loops over each object and calls some methods on them. Every user will have at least one calendar (the default one) and may be subscribed to additional ones.

#### Code Example 9.9

```
1 // Write some basic information for all
2 // calendars available to the user to
3 // the log.
4 function calendarSummary() {
5     var cal =
6         CalendarApp.getAllOwnedCalendars();
7     Logger.log('Number of Calendars: '
8         + cal.length);
9     cal.forEach(
10        function(cal) {
11            Logger.log('Calendar Name: ' +
12                cal.getName());
13            Logger.log('Is primary calendar? ' +
14                cal.isMyPrimaryCalendar());
15            Logger.log('Calendar description: ' +
16                cal.getDescription());
17        });
18    });
19 }
```

## 9.5.2 Add Events To A Calendar Using Spreadsheet Input

The next example shows how a range of dates can be taken from a spreadsheet and be used as input to create a series of all-day events. The range input in cells A1 to A9 was taken from a sheet named “Holidays” and the dates were from August 8th to August 16th 2013. The script example extracts the dates from the spreadsheet cells and uses them to create all day events.

### Code Example 9.10

```
1 // Take a range of dates from 29th September
2 // 2015 to 7th October 2015 (inclusive) from a
3 // spreadsheet and create calendar
4 // all-day events for each of these dates.
5 // The title is set as "Holidays" and the
6 // description to "Forget about work".
7 function calAddEvents() {
8   var cal = CalendarApp.getDefaultCalendar(),
9       ss =
10      SpreadsheetApp.getActiveSpreadsheet(),
11      sh = ss.getSheetByName('Holidays'),
12      holDates =
13      sh.getRange('A1:A9').getValues().map(
14        function (row) {
15          return row[0];
16        });
17      holDates.forEach(
18        function (holDate) {
19          var calEvent;
20          calEvent =
21          cal.createAllDayEvent('Holiday',
22                                holDate);
23          calEvent.setDescription(
```

```
24         'Forget about work')
25     });
26 }
```

Once this code is executed, the events will appear in the calendar view for the relevant dates. Sometimes a calendar “close and reopen” action is required to see the added events. Only the actual dates for the calendar events were taken from the spreadsheet but, with a little more code, additional input could have been specified. The `createEvent()` and `createAllDayEvent()` have various overloads and these allow additional `CalendarEvent` properties such as event location to be specified. All these additional values could be listed in the spreadsheet.

### 9.5.3 Remove Events From A Calendar

Sorry, holidays cancelled! The next example removes all the events created above from the default calendar. Firstly, it gets an array of `CalendarEvent` objects for the indicated date range. It then filters them on the basis of their titles and removes all events with the title “Holiday”.

#### Code Example 9.11

```
1 // Remove all events from the default calendar
2 // for the dates between 29th September
3 // 2015 to 7th October 2015.
4 // inclusive where the event title equals
5 // "Holiday".
6 // First get all events for the date range
7 // as an array.
8 // Then filter this array to
9 // get only those with the indicated title.
10 // Remove those filtered events from the
```

```
11 // calendar.
12 function calRemoveEvents() {
13     var cal = CalendarApp.getDefaultCalendar(),
14         calEvents,
15         eventTitleToCancel = 'Holiday',
16         toCancelEvents = [];
17     calEvents =
18         cal.getEvents(new Date('September 29, 2015'),
19                       new Date('October 7, 2015'));
20     toCancelEvents = calEvents.filter(
21         function (calEvent) {
22             return (calEvent.getTitle()
23                 ===
24                 eventTitleToCancel);
25         });
26     toCancelEvents.forEach(
27         function (eventToCancel) {
28             eventToCancel.deleteEvent();
29         });
30 }
```

The default calendar should now be restored to its pre-holiday state with all the calendar events added earlier now removed. You should experiment with this and the previous example by using your own dates and events to manipulate your own calendar using GAS.

#### 9.5.4 Write A Summary Of Calendar Events For a Given Date Interval To A Spreadsheet

An earlier example showed how input from a spreadsheet could be used to create calendar events. Spreadsheets can also be used to collect information on events already in a calendar. The following example queries the default calendar for event information for the month of September 2015.



**Code Example 9.12**

```
1 // Get the default calendar object.
2 // Insert a new sheet into the active
3 // spreadsheet.
4 // Retrieve all CalendarEvent objects
5 // for a specified date interval
6 // (month of September 2015).
7 // Append a header row to the new sheet
8 // and populate it.
9 // Make the header row font bold.
10 // Loop over the array of CalendarEvent objects
11 // and use their methods to extract property
12 // values of interest.
13 // Write the property values to the new sheet.
14 function writeCalEventsToSheet() {
15     var cal = CalendarApp.getDefaultCalendar(),
16         ss =
17         SpreadsheetApp.getActiveSpreadsheet(),
18         newShName = '',
19         newSh = ss.insertSheet(newShName),
20         startDate = new Date('September 01, 2015'),
21         endDate = new Date('October 01, 2015'),
22         calEvents = cal.getEvents(startDate, endDate),
23         colHeaders = [],
24         colCount,
25         colHeaderRng;
26     colHeaders = ['Title',
27                 'Description',
28                 'EventId',
29                 'DateCreated',
30                 'IsAllDay',
31                 'MyStatus',
32                 'StartTime',
33                 'EndTime',
```

```

34         'Location'];
35     colCount = colHeaders.length;
36     newSh.appendRow(colHeaders);
37     colHeaderRng = newSh.getRange(1,
38         1,
39         1,
40         colCount);
41     colHeaderRng.setFontWeight('bold');
42     calEvents.forEach(
43         function (calEvent) {
44             newSh.appendRow(
45                 [calEvent.getTitle(),
46                 calEvent.getDescription(),
47                 calEvent.getId(),
48                 calEvent.getDateCreated(),
49                 calEvent.isAllDayEvent(),
50                 calEvent.getMyStatus(),
51                 calEvent.getStartTime(),
52                 calEvent.getEndTime(),
53                 calEvent.getLocation()
54             ]);
55         });
56 }

```

Executing function *writeCalEventsToSheet()* example 9.12 should add a new sheet to your active spreadsheet and populate it with details for events for the month of September 2015 (assuming your calendar has some events!). If you add new events and attempt to re-run the function, remember to delete the old sheet named “CalendarEvents” first to avoid an error. The code comments should suffice to explain what this function achieves. There are nine method calls on the *CalendarEvent* object within the *forEach* loop. Most of these have *set* method equivalents. One all important, and by now familiar, method is *getId()*. As with *File*, *Folder*, *GmailThread*, and so on, many objects have these system-assigned IDs that are not

useful for the end-user but are very useful for the developer. These IDs uniquely identify the object and many objects have methods to retrieve objects by ID. This code example has only shown a subset of the *CalendarEvent* methods that are available and there are others that are well worth investigating.

## 9.6 Summary

- The *MailApp* object functionality is limited but it can send emails.
- Users are restricted to sending a maximum of 100 mails per day and the number of recipients in each sent mail counts towards this maximum.
- The *GmailApp* object is much more powerful than *MailApp*.
- Only a limited set of file types can be sent as attachments using either *MailApp* or *GmailApp* objects.
- *GmailThread* objects can be retrieved using a variety of *GmailApp* methods.
- Email threads contain one or more email messages (*GmailMessage* objects) and all email messages belong to a thread.
- Both *GmailThread* and *GmailMessage* objects possess a large number of methods for getting and setting their properties.
- Labels can be added to email threads and are very useful for flagging and filtering the threads to which they are attached.
- Gmail calendars are manipulated via the *CalendarApp* object.
- All Gmail accounts have a default calendar and may subscribe to additional calendars.
- Calendar appointments (called events) are represented by *CalendarEvent* objects.
- Check the [Google documentation](https://developers.google.com/apps-script/)<sup>66</sup> for up-to-date information on all the objects discussed in this chapter.

---

<sup>66</sup><https://developers.google.com/apps-script/>

# Appendix A: Excel VBA And Google Apps Script Comparison

## Introduction

Microsoft Excel™ remains the dominant spreadsheet application so many of those coming to Google Apps Script programming will be very familiar with it. It hosts a programming language called Visual Basic for Applications™ (VBA) that can be used to extend functionality, build user interfaces, integrate with other Microsoft Office™ applications, and as a front end to relational databases. This appendix aims to provide a quick reference for Excel VBA programmers by giving some VBA examples in parallel with equivalent Google Apps Script (basically JavaScript) code for some common spreadsheet programming tasks. VBA and Google Apps Script are very different languages and the APIs they each use for spreadsheet programming have been developed independently so their respective methods and properties also differ. Despite these differences, the objectives of both languages and the tasks they are applied to are similar to the extent that an experienced VBA programmer should be able to pick up Google Apps Script quite quickly. This appendix assumes knowledge of VBA and aims to facilitate the reader's transition to the Google Apps Script environment.

The examples below are given in pairs: First the VBA code and then the **functionally equivalent** Google Apps Script version with some explanatory comments. The VBA is generally not commented because the examples assume VBA expertise. Comments are added,

however, for trickier and longer examples or when the code examples in the two languages are very different due to inherent VBA/JavaScript differences or divergent API approaches. The code examples should perform as described but in many instances alternative approaches can be used and the examples may not be optimal.

The VBA examples given generally write their output to the **Immediate Window** while the Google Apps Script equivalents write to the **Logger**. Some examples write to or format spreadsheet cells.

### Google Apps Script Or JavaScript

“JavaScript” is used here to refer to general JavaScript concepts such as arrays.

Google Apps Script refers to the specific Google implementation as it applies to spreadsheet programming and Google App Script APIs. The context should make the usage clear.

## Spreadsheets and Sheets

Spreadsheets and sheets are handled similarly. One difference of note is that Google Apps Script does not make a distinction between *Sheets* and *Worksheets* as VBA does.

### Active Spreadsheet

Multiple spreadsheet files can be open at a given time but only one is *active*.

## VBA

```
1 Public Sub SpreadsheetInstance()  
2     Dim ss As Workbook  
3     Set ss = Application.ActiveWorkbook  
4     Debug.Print ss.Name  
5 End Sub
```

## Google Apps Script

```
1 function spreadsheetInstance() {  
2     var ss = SpreadsheetApp.getActiveSpreadsheet();  
3     Logger.log(ss.getName());  
4 }
```

VBA uses the *Name* property while Google Apps Script uses the method *getName()* to return the value.

## Sheet/Worksheet

Spreadsheets contain sheets. In VBA these are stored as *collections* and in Google Apps Script as JavaScript arrays of *Sheet* objects. The pairs of examples given here call some *Sheet* methods and print the output.

## VBA

```
1 Public Sub FirstSheetInfo()  
2     Dim sh1 As Worksheet  
3     Set sh1 = ActiveWorkbook.Worksheets(1)  
4     Dim usedRng As Range  
5     Set usedRng = sh1.UsedRange  
6     Debug.Print sh1.Name  
7     Debug.Print usedRng.Address  
8 End Sub
```

## Google Apps Script

```
1 function firstSheetInfo() {  
2     var ss = SpreadsheetApp.getActiveSpreadsheet(),  
3         sheets = ss.getSheets(),  
4         // getSheets() returns an array  
5         // JavaScript arrays are always zero-based  
6         sh1 = sheets[0];  
7     Logger.log(sh1.getName());  
8     // getDataRange is analagous to UsedRange  
9     // in VBA  
10    // getA1Notation() is functional equivalent to  
11    // Address in VBA  
12    Logger.log(sh1.getDataRange().getA1Notation());  
13 }
```

## Sheet Collections

The previous examples extracted a single *Sheet* object and called some of its methods. The example pairs here loop over the all the sheets of the active spreadsheet and print the sheet names.

## VBA

```
1 Public Sub PrintSheetNames()  
2     Dim sheets As Worksheets  
3     Dim sheet As Worksheet  
4     For Each sheet In ActiveWorkbook.Sheets  
5         Debug.Print sheet.Name  
6     Next sheet  
7 End Sub
```

## Google Apps Script

```
1 // Print the names of all sheets in the active  
2 // spreadsheet.  
3 function printSheetNames() {  
4     var ss = SpreadsheetApp.getActiveSpreadsheet(),  
5         sheets = ss.getSheets(),  
6         i;  
7     for (i = 0; i < sheets.length; i += 1) {  
8         Logger.log(sheets[i].getName());  
9     }  
10 }
```

## Adding And Removing Sheets

Spreadsheet applications may need to add new sheets to an existing spreadsheet file and then, after some processing, they may need to then remove one or more sheets. Both tasks are easily achieved in both VBA and and Google Apps Script.

### VBA



```
1 ' Add a new sheet to a workbook.
2 ' Call the Add method of the
3 ' Worksheets collection
4 ' Assign a name to the returned
5 ' Worksheet instance
6 ' Name property.
7 Sub AddNewSheet()
8     Dim newSheet As Worksheet
9     Set newSheet = ActiveWorkbook.Worksheets.Add
10    newSheet.Name = "AddedSheet"
11    MsgBox "New Sheet Added!"
12 End Sub
13
14 ' Delete a named sheet from the
15 ' active spreadsheet.
16 ' The sheet to delete is identified
17 ' in the Worksheets collection
18 ' by name. The returned instance
19 ' is deleted by calling its
20 ' Delete method.
21 ' MS Excel will prompt to confirm.
22 Sub RemoveSheet()
23     Dim sheetToRemove As Worksheet
24     Set sheetToRemove = _
25     ActiveWorkbook.Worksheets("AddedSheet")
26     sheetToRemove.Delete
27     MsgBox "Sheet Deleted!"
28 End Sub
```

## Google Apps Script

```
1 // Add a new sheet to the active spreadsheet.
2 // Get an instance of the active spreadsheet.
3 // Call its insertSheet method.
4 // Call the setName method of the
5 // returned instance.
6 function addNewSheet() {
7     var ss =
8         SpreadsheetApp.getActiveSpreadsheet(),
9         newSheet;
10    newSheet = ss.insertSheet();
11    newSheet.setName("AddedSheet");
12    Browser.msgBox("New Sheet Added!");
13 }
14
15 // Remove a named sheet from the
16 // active spreadsheet.
17 // Get an instance of the active
18 // spreadsheet.
19 // Get an instance of the sheet to remove.
20 // Activate the sheet to remove
21 // Call the spreadsheet instance method
22 // deleteActiveSheet.
23 function removeSheet() {
24     var ss =
25         SpreadsheetApp.getActiveSpreadsheet(),
26         sheetToRemove =
27             ss.getSheetByName("AddedSheet");
28    sheetToRemove.activate();
29    ss.deleteActiveSheet();
30    Browser.msgBox("SheetDeleted!");
31 }
```

The code comments in both languages should adequately describe the actions and objects required to add and remove sheets from both spreadsheet applications. The Google Apps Script mechanism

appears a little more complicated than its VBA equivalent. In order to remove a sheet, it first has to be activated so that the *Spreadsheet* instance method *deleteActiveSheet()* can be called. Otherwise, both languages work quite similarly.

## Hiding And Unhiding Sheets

Hiding sheets can help to keep a spreadsheet uncluttered and easy to use while also helping to prevent inadvertent changes to important data. Lists of values that the application uses may not be important to the users so they can be hidden from their view while still remaining available to the application code. The VBA and Google Apps Script code required to do the hiding, unhiding and listing of hidden sheets is very similar.

**Hiding a sheet identified by name.**

### VBA

```
1 Public Sub SheetHide()  
2     Dim sh As Worksheet  
3     Set sh = Worksheets.Item("ToHide")  
4     sh.Visible = False  
5 End Sub
```

### Google Apps Script

```
1 // Hide a sheet specified by its name.
2 function sheetHide() {
3   var ss =
4     SpreadsheetApp.getActiveSpreadsheet(),
5     sh = ss.getSheetByName('ToHide');
6   sh.hideSheet()
7 }
```

### Listing hidden sheets

#### VBA

```
1 Public Sub ListHiddenSheetNames()
2   Dim sheet As Worksheet
3   For Each sheet In Worksheets
4     If sheet.Visible = False Then
5       Debug.Print sheet.Name
6     End If
7   Next sheet
8 End Sub
```

#### Google Apps Script

```
1 // Write a list of hidden sheet names to log.
2 function listHiddenSheetNames() {
3   var ss =
4     SpreadsheetApp.getActiveSpreadsheet(),
5     sheets = ss.getSheets();
6   sheets.forEach(
7     function (sheet) {
8       if (sheet.isSheetHidden()) {
9         Logger.log(sheet.getName());
10      }
11    });
12 }
```

## Unhiding hidden sheets

### VBA

```
1 Public Sub SheetsUnhide()  
2     Dim sheet As Worksheet  
3     For Each sheet In Worksheets  
4         If sheet.Visible = False Then  
5             sheet.Visible = True  
6         End If  
7     Next sheet  
8 End Sub
```

### Google Apps Script

```
1 // Unhide all hidden sheets.  
2 function sheetsUnhide() {  
3     var ss =  
4         SpreadsheetApp.getActiveSpreadsheet(),  
5         sheets = ss.getSheets();  
6     sheets.forEach(  
7         function (sheet) {  
8             if (sheet.isSheetHidden()) {  
9                 sheet.showSheet();  
10            }  
11        });  
12 }
```

The main difference in the approach taken by each language in these examples is how they iterate over the *Worksheets* collection in VBA and the array of *Sheet* objects in Google Apps Script. Newer versions of JavaScript, including Google Apps Script, have added some very powerful methods to arrays. Included in these are methods that take a callback function as an argument that is invoked for each element in the array. The *forEach()* method above

is an example. It operates in a similar manner to the VBA *For Each* loop but unlike it, *forEach()* needs a function as an argument. In the examples above anonymous functions were used. This type of approach where functions take other functions as arguments is very powerful but may be unfamiliar to VBA programmers.

## Protecting Sheets

Hiding sheets provides a type of “security through obscurity” but does not prevent deliberate tampering. Both VBA and Google Apps Script allow you to protect individual worksheets within a spreadsheet but they take very approaches to this.

### VBA

```
1 ' Password-protect rotect a sheet identified
2 ' by name
3 Public Sub SheetProtect()
4     Dim sh As Worksheet
5     Dim pwd As String: pwd = "secret"
6     Set sh = Worksheets.Item("ToProtect")
7     sh.Protect pwd
8 End Sub
```

### Google Apps Script

```
1 // Identify a sheet by name to protect
2 // When this code runs, the lock icon
3 // will appear on the sheet name.
4 // Share the spreadsheet with another user
5 // as an editor. That user can edit all
6 // sheets except the protected one. The user
7 // can still edit the protected sheet.
8 function sheetProtect() {
9     var ss =
10         SpreadsheetApp.getActiveSpreadsheet(),
11         sh = ss.getSheetByName('ToProtect'),
12         permissions = sh.getSheetProtection();
13     ss.addEditor(<gmail address goes here>);
14     permissions.setProtected(true);
15     sh.setSheetProtection(permissions);
16 }
```

In VBA, a password is set and the protected is using the *Worksheet Protect* method passing it the password string as an argument. Once protected even the spreadsheet file owner needs to know the password to do anything to the sheet. Google Apps Script takes a different approach. By default, only the file creator can see or edit the spreadsheet. The owner can then add editors or viewers to the spreadsheet. A viewer can see all the sheets but not edit them while the editor can, as the name suggests, edit the sheet contents. However, a single sheet can be protected so that it can be viewed by a user granted editor privilege but is not editable by them. The code example given above shows how this can be done. Unlike in VBA, however, the owner of the spreadsheet will always have full edit permissions on all sheets. In other words, the owner cannot remove permissions from themselves.

## Ranges

Spreadsheet programming is largely about manipulating ranges so this is a long section.

## Selection

Requiring a user to select an input range is a common feature of spreadsheet applications. In order to process the selected cells, the application needs to determine:

- The sheet containing the selection
- The location of the selection within the sheet as given by its address
- The dimensions of the selection, that is the number of rows and columns in the selection

This information is extracted and printed in the following examples

### VBA

```
1 Public Sub PrintSelectionDetails()  
2     Debug.Print "Selected Range Details: "  
3     Debug.Print "-- Sheet: " & _  
4         Selection.Worksheet.Name  
5     Debug.Print "-- Address: " & _  
6         Selection.Address  
7     Debug.Print "-- Row Count: " & _  
8         Selection.Rows.Count  
9     Debug.Print "'-- Column Count: " & _  
10        Selection.Columns.Count  
11 End Sub
```

### Google Apps Script



```
1 // Prints details about selected range in
2 // active spreadsheet
3 // To run, paste code into script editor,
4 // select some cells on any sheet,
5 // execute code and
6 // check log to see details
7 // Prints details about selected range
8 // in active spreadsheet
9 // To run, paste code into script editor,
10 // select some cells on any sheet,
11 // execute code and
12 // check log to see details
13 function printSelectionDetails() {
14     var ss =
15         SpreadsheetApp.getActiveSpreadsheet(),
16         selectedRng = ss.getActiveRange();
17     Logger.log('Selected Range Details:');
18     Logger.log('-- Sheet: '
19         + selectedRng
20         .getSheet()
21         .getSheetName());
22     Logger.log('-- Address: '
23         + selectedRng.getA1Notation());
24     Logger.log('-- Row Count: '
25         + ((selectedRng.getLastRow() + 1)
26         - selectedRng.getRow()));
27     Logger.log('-- Column Count: '
28         + ((selectedRng.getLastColumn() + 1)
29         - selectedRng.getColumn()));
30 }
```

VBA provides the handy *Selection* object which is of type *Range* and its methods can be used to extract the required information. The Google Apps Script *Spreadsheet* object provides the *getActiveSelection()* method to return the Google Spreadsheets equivalent to

the VBA *Selection*. Its *getRow()* and *getColumn()* methods return the row number of the first row and first column, respectively, for the *Range* object on which they are invoked. The purpose of the *getLastRow()* and *getLastColumn()* *Range* methods is clear from their names. By using a combination of these methods the VBA *Selection.Rows.Count* and *Selection.Columns.Count* properties can be mimicked as was done above.

## Used Range

To retrieve the very useful equivalent of the VBA *UsedRange* object in Google Apps Script, use the *Sheet getDataRange()* method. In both languages it is easy to transfer the cell contents of a range into an array. JavaScript arrays are a lot more flexible than those in VBA and they are always zero-based. JavaScript's dynamic typing also makes matters more straightforward. VBA is a typed language but its *Variant* type negates the all the type-checking. However, it has to be used to receive the *Range value* property. Another fundamental language difference is that JavaScript does not distinguish functions and subroutines. Instead functions are always used and if there is no explicit *return* statement, *undefined* is the return value.

## VBA

```
1 Public Function GetUsedRangeAsArray(sheetName _  
2                               As String) As Variant  
3     Dim sh As Worksheet  
4     Set sh = _  
5         ActiveWorkbook.Worksheets(sheetName)  
6     GetUsedRangeAsArray = sh.UsedRange.value  
7 End Function  
8 Sub test_GetUsedRangeAsArray()  
9     Dim sheetName As String  
10    Dim rngValues
```

```
11     Dim firstRow As Variant
12     sheetName = "Sheet1"
13     rngValues = GetUsedRangeAsArray(sheetName)
14     Debug.Print rngValues(1, 1)
15     Debug.Print UBound(rngValues)
16     Debug.Print UBound(rngValues, 2)
17 End Sub
```

## Google Apps Script

```
1  function getUsedRangeAsArray(sheetName) {
2      var ss =
3          SpreadsheetApp.getActiveSpreadsheet(),
4          sh = ss.getSheetByName(sheetName);
5      // The getValues() method of the
6      // Range object returns an array of arrays
7      return sh.getDataRange().getValues();
8  }
9  // JavaScript does not distinguish between
10 // subroutines and functions.
11 // When the return statement is omitted,
12 // functions return undefined.
13 function test_getUsedRangeAsArray() {
14     var ss = SpreadsheetApp.getActiveSpreadsheet(),
15         sheetName = 'Sheet1',
16         rngValues = getUsedRangeAsArray(sheetName);
17     // Print the number of rows in the range
18     // The toString() call to suppress the
19     // decimal point so
20     // that, for example, 10.0, is reported as 10
21     Logger.log((rngValues.length).toString());
22     // Print the number of columns
23     // The column count will be the same
24     // for all rows so only need the first row
25     Logger.log((rngValues[0].length).toString());
```

```

26 // Print the value in the first cell
27 Logger.log(rngValues[0][0]);
28 }

```

## Add Colours To Range In First Sheet

Cells and their contents can be programmatically formatted just as easily in Google Spreadsheets as in Excel.

### VBA

```

1 Sub AddColorsToRange()
2     Dim sh1 As Worksheet
3     Dim addr As String: addr = "A4:B10"
4     Set sh1 = ActiveWorkbook.Worksheets(1)
5     sh1.Range(addr).Interior.ColorIndex = 3
6     sh1.Range(addr).Font.ColorIndex = 10
7 End Sub

```

### Google Apps Script

```

1 // Select a block of cells in the first sheet.
2 // Use Range methods to set both the font and
3 // background colors.
4 function addColorsToRange() {
5     var ss =
6         SpreadsheetApp.getActiveSpreadsheet(),
7         sheets = ss.getSheets(),
8         sh1 = sheets[0],
9         addr = 'A4:B10',
10        rng;
11 // getRange is overloaded. This method can
12 // also accept row and column integers
13 rng = sh1.getRange(addr);

```

```
14   rng.setFontColor('green');
15   rng.setBackgroundColor('red');
16 }
```

## Range Offsets

The *offset Range* property in VBA is implemented in Google Apps Script as the *Range offset()* method. In its basic form, the Google Apps Script version can be used to exactly mimic its VBA namesake as the following code demonstrates.

### VBA

```
1  Public Sub OffsetDemo()
2      Dim sh As Worksheet
3      Dim cell As Range
4      Set sh = _
5          ActiveWorkbook.Worksheets(1)
6      Set cell = sh.Range("B2")
7      cell.value = "Middle"
8      cell.Offset(-1, -1).value = "Top Left"
9      cell.Offset(0, -1).value = "Left"
10     cell.Offset(1, -1).value = "Bottom Left"
11     cell.Offset(-1, 0).value = "Top"
12     cell.Offset(1, 0).value = "Bottom"
13     cell.Offset(-1, 1).value = "Top Right"
14     cell.Offset(0, 1).value = "Right"
15     cell.Offset(1, 1).value = "Bottom Right"
16 End Sub
```

### Google Apps Script

```
1 // The Spreadsheet method getSheets() returns
2 // an array.
3 // The code "ss.getSheets()[0]"
4 // returns the first sheet and is equivalent to
5 // "ActiveWorkbook.Worksheets(1)" in VBA.
6 // Note that the VBA version is 1-based!
7 function offsetDemo() {
8     var ss =
9         SpreadsheetApp.getActiveSpreadsheet(),
10        sh = ss.getSheets()[0],
11        cell = sh.getRange('B2');
12    cell.setValue('Middle');
13    cell.offset(-1,-1).setValue('Top Left');
14    cell.offset(0, -1).setValue('Left');
15    cell.offset(1, -1).setValue('Bottom Left');
16    cell.offset(-1, 0).setValue('Top');
17    cell.offset(1, 0).setValue('Bottom');
18    cell.offset(-1, 1).setValue('Top Right');
19    cell.offset(0, 1).setValue('Right');
20    cell.offset(1, 1).setValue('Bottom Right');
21 }
```

Pasting and executing these code snippets in either spreadsheet application writes the location of cell B2's neighbours relative to its location. The Google Apps Script *offset()* method is, however, **overloaded**. This concept was discussed in chapter 5 in relation to the *Sheet getRange()* method but it merits re-visiting here to show how the functionality of its overloaded versions can be implemented in VBA.

## VBA

```

1  ' Mimicking Google Apps Script
2  ' offset() method overloads.
3  Public Sub OffsetOverloadDemo()
4      Dim sh As Worksheet
5      Dim cell As Range
6      Dim offsetRng2 As Range
7      Dim offsetRng3 As Range
8      Set sh = ActiveWorkbook.Worksheets(1)
9      Set cell = sh.Range("A1")
10     'Offset returns a Range so Offset
11     ' can be called again
12     ' on the returned Range from
13     ' first Offset call.
14     Set offsetRng2 = Range(cell.Offset(1, 4), _
15         cell.Offset(1, 4).Offset(1, 0))
16     Set offsetRng3 = Range(cell.Offset(10, 4), _
17         cell.Offset(10, 4).Offset(3, 4))
18     Debug.Print offsetRng2.Address
19     Debug.Print offsetRng3.Address
20 End Sub

```

## Google Apps Script

```

1  // Demonstrating overloaded versions of offset()
2  // Output:
3  // Address of offset() overload 2
4  // (rowOffset, columnOffset, numRows) is: E2:E3
5  // Address of offset() overload 3 (rowOffset,
6  //   columnOffset, numRows, numColumns)
7  //   is: E11:I14
8  function offsetOverloadDemo() {
9      var ss =
10         SpreadsheetApp.getActiveSpreadsheet(),
11         sh = ss.getSheets()[0],
12         cell = sh.getRange('A1'),

```

```
13     offsetRng2 = cell.offset(1, 4, 2),
14     offsetRng3 = cell.offset(10, 4, 4, 5);
15     Logger.log('Address of offset() overload 2 ' +
16         '(rowOffset, columnOffset, numRows) is: ' +
17         + offsetRng2.getA1Notation());
18     Logger.log('Address of offset() overload 3 ' +
19         '(rowOffset, columnOffset, numRows, ' +
20         'numColumns) is: ' +
21         + offsetRng3.getA1Notation());
22 }
```

While the VBA version defines the same ranges as the Google Apps Script version, it is not exactly clear. The key point to realise is that the VBA *Range Offset* property returns another *Range* so there is no reason why *Offset* cannot be invoked again on this returned *Range*. However, code like this VBA example should be avoided where possible and, if it cannot be avoided, it had better be well commented and documented! It was given here purely for demonstration purposes.

## Named Ranges

The advantages of using named ranges were outlined in chapter 5. Google Apps Script provides *Spreadsheet* methods for setting named ranges and for retrieving the *Range* objects that the names refer to, see chapter 5 for a full discussion. However, there does not appear to be a way to implement the following VBA functionality.

### VBA



```
1 Public Sub PrintRangeNames()  
2     Dim namedRng As Name  
3     For Each namedRng In ActiveWorkbook.Names  
4         Debug.Print "The name of the range is: " & _  
5             namedRng.Name & _  
6             " It refers to this address: " & _  
7                 namedRng.RefersTo  
8     Next namedRng  
9 End Sub
```

This VBA code prints details for all named ranges in the active Excel file. This functionality can be very useful but at the time of writing, I was unable to duplicate it in Google Apps Script.

## Cell Comments

Cell comments are a good way to document spreadsheets and add useful metadata that can describe the meaning of cell contents. They are also amenable to programmatic manipulation.

The Google Apps Script equivalent to Excel comments are **notes**. These are *Range* attributes that can be set and retrieved with with the *Range* getters and setters *setNote()* and *getNote()*, respectively.



## Cell Comments

Comments in Google Spreadsheets set from the spreadsheet are **not** the same as notes set programmatically. There does not appear to be a way to programmatically manipulate comments set from the spreadsheet by users.

## Setting Cell Comments

### VBA

```
1 Public Sub SetCellComment(sheetName As String, _
2                             cellAddress As String, _
3                             cellComment As String)
4     Dim sh As Worksheet
5     Dim cell As Range
6     Set sh = ActiveWorkbook.Worksheets(sheetName)
7     Set cell = sh.Range(cellAddress)
8     cell.AddComment cellComment
9 End Sub
10 Public Sub test_SetCellComment()
11     Dim sheetName As String
12     sheetName = "Sheet1"
13     Dim cellAddress As String
14     cellAddress = "C10"
15     Dim cellComment As String
16     cellComment = "Comment added: " & Now()
17     Call SetCellComment(sheetName, _
18                         cellAddress, _
19                         cellComment)
20 End Sub
```

## Google Apps Script

```
1 function setCellComment(sheetName, cellAddress,
2                             cellComment) {
3     var ss =
4         SpreadsheetApp.getActiveSpreadsheet(),
5         sh = ss.getSheetByName(sheetName),
6         cell = sh.getRange(cellAddress);
7     cell.setNote(cellComment);
8 }
9 function test_setCellComment() {
10     var sheetName = 'Sheet1',
11         cellAddress = 'C10',
12         cellComment = 'Comment added ' + Date();
```

```
13 setCellComment(sheetName, cellAddress, cellComment);
14 }
```

## Removing Cell Comments

### VBA

```
1 ' Need to check if the cell has a comment.
2 ' If it does not, then exit the sub but if
3 ' it does, then remove it.
4 Public Sub RemoveCellComment(sheetName _
5                               As String, _
6                               cellAddress As String)
7     Dim sh As Worksheet
8     Dim cell As Range
9     Set sh = ActiveWorkbook.Worksheets(sheetName)
10    Set cell = sh.Range(cellAddress)
11    If cell.Comment Is Nothing Then
12        Exit Sub
13    Else
14        cell.Comment.Delete
15    End If
16 End Sub
17 Public Sub test_RemoveCellComment()
18     Dim sheetName As String
19     sheetName = "Sheet1"
20     Dim cellAddress As String
21     cellAddress = "C10"
22     Call RemoveCellComment(sheetName, _
23                             cellAddress)
24 End Sub
```

### Google Apps Script

```
1 // To remove a comment, just pass an empty string
2 // to the setNote() method.
3 function removeCellComment(sheetName, cellAddress) {
4     var ss =
5         SpreadsheetApp.getActiveSpreadsheet(),
6         sh = ss.getSheetByName(sheetName),
7         cell = sh.getRange(cellAddress);
8     cell.setNote('');
9 }
10 function test_removeCellComment() {
11     var sheetName = 'Sheet1',
12         cellAddress = 'C10';
13     removeCellComment(sheetName, cellAddress);
14 }
```

## Selectively Copy Rows From One Sheet To A New Sheet

Copying rows from one sheet to another based on some pre-determined criterion is a common spreadsheet task. Given the input in the figure below, the code examples given do the following:

- Insert a new sheet named “Target” into which rows will be copied
- Copy the header row to the new sheet
- Check each of the data rows and if the second column value is less than or equal to 10000 then copy the row to the new sheet.

	Name	
	A	B
1	<b>Name</b>	<b>Salary</b>
2	Jones	24635
3	Calvez	6568
4	Smith	16750
5	Patel	61792
6	Lee	35077
7	Myers	8279
8	Agnew	37625
9	Novak	43665
10	Murphy	9378
11		

Figure Appendix A.1: Data input sheet

## VBA

```

1  ' This VBA code is commented because the
2  ' VBA approach differs
3  ' considerably from the Google Apps Script one.
4  ' Note: the Offset() method of the Range
5  ' object uses 0-based indexes.
6  Public Sub copyRowsToNewSheet()
7      Dim sourceSheet As Worksheet
8      Dim newSheet As Worksheet
9      Dim newSheetName As String
10     newSheetName = "Target"
11     Dim sourceRng As Range
12     Dim sourceRows As Variant
13     Dim i As Long
14     Set sourceSheet = _
15         Application.Worksheets("Source")
16     Set newSheet = ActiveWorkbook.Worksheets.Add
17     newSheet.Name = newSheetName
18     ' Use a named range as marker
19     ' for row copying (VBA hack!)
20     newSheet.Range("A1").Name = "nextRow"

```

```
21 Set sourceRng = sourceSheet.UsedRange
22 ' Copy the header row
23 sourceRng.Rows(1).Copy Range("nextRow")
24 ' Moved the named range marker down one row
25 Range("nextRow").Offset(1, 0).Name = _
26     "nextRow"
27 'Skip header row by setting i,
28 ' the row counter, = 2
29 ' i starts at 2 to skip header row
30 For i = 2 To sourceRng.Rows.Count
31     If sourceRng.Cells(i, 2).value _
32         <= 10000 Then
33         ' Define the row range to copy
34         ' using the first and
35         ' last cell in the row.
36         Range(sourceRng.Cells(i, 1), _
37             sourceRng.Cells(i, _
38                 sourceRng.Columns.Count)).Copy _
39             Range("nextRow")
40         Range("nextRow").Offset(1, 0).Name _
41             = "nextRow"
42     End If
43 Next i
44 End Sub
```

## Google Apps Script

```

1 // Longer example
2 // Copy rows from one sheet named "Source" to
3 // a newly inserted
4 // one based on a criterion check of second
5 // column.
6 // Copy the header row to the new sheet.
7 // If Salary <= 10,000 then copy the entire row
8 function copyRowsToNewSheet() {
9   var ss =
10     SpreadsheetApp.getActiveSpreadsheet(),
11     sourceSheet = ss.getSheetByName('Source'),
12     newSheetName = 'Target',
13     newSheet = ss.insertSheet(newSheetName),
14     sourceRng = sourceSheet.getDataRange(),
15     sourceRows = sourceRng.getValues(),
16     i;
17   newSheet.appendRow(sourceRows[0]);
18   for (i = 1; i < sourceRows.length; i += 1) {
19     if (sourceRows[i][1] <= 10000) {
20       newSheet.appendRow(sourceRows[i]);
21     }
22   }
23 }

```

The output from these code examples is shown below.

	A	B
1	Name	Salary
2	Calvez	6568
3	Myers	8279
4	Murphy	9378
5		

Figure Appendix A.1: Data output sheet

The Google Apps Script *Sheet appendRow()* is very convenient and significantly simplifies the code when compared to the VBA

example. Taking an array of values, it just adds a row to the sheet that contains these values. In VBA the range name “next” is used as a marker for the destination to where the selected row is copied. After each copying operation, it has to be moved down by one row to be ready for the next row to copy.

## Print Addresses And Formulas For Range

The code examples below demonstrate how to loop over a range of cells one cell at a time.

### VBA

```
1 Public Sub test_PrintSheetFormulas()  
2     Dim sheetName As String  
3     sheetName = "Formulas"  
4     Call PrintSheetFormulas(sheetName)  
5 End Sub  
6 Public Sub PrintSheetFormulas(sheetName _  
7                               As String)  
8     Dim sourceSheet As Worksheet  
9     Dim usedRng As Range  
10    Dim i As Long  
11    Dim j As Long  
12    Dim cellAddr As String  
13    Dim cellFormula As String  
14    Set sourceSheet = _  
15        ActiveWorkbook.Worksheets(sheetName)  
16    Set usedRng = sourceSheet.UsedRange  
17    For i = 1 To usedRng.Rows.Count  
18        For j = 1 To usedRng.Columns.Count  
19            cellAddr = _  
20                usedRng.Cells(i, j).Address  
21            cellFormula = _
```



```

22         usedRng.Cells(i, j).Formula
23         If Left(cellFormula, 1) = "=" Then
24             Debug.Print cellAddr & _
25                 ": " & cellFormula
26         End If
27     Next j
28 Next i
29 End Sub

```

## Google Apps Script

```

1  function test_printSheetFormulas() {
2      var sheetName = 'Formulas';
3      printSheetFormulas(sheetName);
4  }
5  function printSheetFormulas(sheetName) {
6      var ss =
7          SpreadsheetApp.getActiveSpreadsheet(),
8          sourceSheet = ss.getSheetByName(sheetName),
9          usedRng = sourceSheet.getDataRange(),
10         i,
11         j,
12         cell,
13         cellAddr,
14         cellFormula;
15     for (i = 1; i <= usedRng.getLastRow();
16         i += 1) {
17         for (j = 1; j <= usedRng.getLastColumn();
18             j += 1) {
19             cell = usedRng.getCell(i, j);
20             cellAddr = cell.getA1Notation();
21             cellFormula = cell.getFormula();
22             if (cellFormula) {
23                 Logger.log(cellAddr +
24                     ': ' + cellFormula);

```

```
25     }  
26   }  
27 }  
28 }
```

The Google Apps Script *Range* *getCell()* method is analogous to the VBA *Range* *Cells* property. Both expect two integer arguments for the row and column indexes and both are one-based.

# Appendix B: Final Notes

## Additional Resources

There are lots of resources for learning JavaScript. Many of these are on-line and free. The focus is generally on JavaScript for web client programming but the core language is the same.

### JavaScript Websites

Here are a few that I have found useful.

- [The W3C School website](#)<sup>67</sup>
- *Eloquent JavaScript* by Marijn Haverbeke. Also available as a book<sup>68</sup>
- [The Mozilla MDN site](#)<sup>69</sup>
- Check out [Stackoverflow entries for Google Spreadsheets](#)<sup>70</sup>
- [Bruce McPherson's](#)<sup>71</sup> site contains lots of good material on Google Apps Script and Excel VBA among lots of other useful stuff.

These are just the books that I have read and used. There are many others.

---

<sup>67</sup><http://www.w3schools.com/js/default.asp>

<sup>68</sup><http://eloquentjavascript.net/contents.html>

<sup>69</sup><https://developers.google.com/apps-script/overview>

<sup>70</sup><http://stackoverflow.com/tags/google-spreadsheet/info>

<sup>71</sup><http://ramblings.mcpher.com/Home/excelquirks>

## JavaScript Books

- *The JavaScript Pocket Guide* by Lenny Burdette (Peachpit Press)
- *JavaScript: The Definitive Guide* 6th Edition by David Flanagan. (O'Reilly)
- *Professional JavaScript for Web Developers* 3rd Edition by Nicholas C. Zakas. (Wrox)
- *JavaScript: The Good Parts* by Douglas Crockford. (O'Reilly)

## Google Apps Script

- *Google Script: Enterprise Application Essentials: Adding Functionality to Your Google Apps* by James Ferreira (O'Reilly)

The Ferreira book is the only one that I have found that is devoted to Google Apps Script. Its emphasis differs from this book in that it does not go into spreadsheet programming in any detail. However, the web application side is well covered so it is definitely worth reading.

The main source of information on Google Apps Script is the [Google documentation](#).<sup>72</sup>

## JSLint

[JSLint](#)<sup>73</sup> the JavaScript code quality tool is an excellent means of checking source code and it also works for Google Apps Script. I wrote a blog entry on its use for quality checking my Google Apps Script. I highly recommend JSLint; It will allow you to write better, more professional code.

---

<sup>72</sup><https://developers.google.com/apps-script/overview>

<sup>73</sup><http://www.jshint.com/>

## Getting Source code For This Book From Github

I have placed all the example code in this book in my [Github repository](#)<sup>74</sup> under the username “Rotifer”, see [link](#).<sup>75</sup>

### Blog Updates

I maintain a blog on Google Spreadsheet programming and Google Apps Script. It is called [Google Spreadsheet Programming With JavaScript](#).<sup>76</sup> It is worth keeping an eye on this blog because topics of interest to readers of this book will be posted here from time to time.

---

<sup>74</sup><https://github.com/Rotifer>

<sup>75</sup>[https://github.com/Rotifer/JavaScript/blob/master/book\\_code.gs](https://github.com/Rotifer/JavaScript/blob/master/book_code.gs)

<sup>76</sup><http://www.javascript-spreadsheet-programming.com>