

Compiled Content

Module 3

MScFE 670

Data Feeds and Technology

```

    $this->repo_path = $repo_path;
    ($repo_path."/config");if ($parse_ini['bare']) {$this->repo_path = $repo_path;$this->
    path = $repo_path;if ($_init) {$this->run('init');}} else {throw new Exception('"' . $r
    * new Exception('"' . $repo_path . '" is not a directory');}} else {if ($create_new) {if
    )) {mkdir($repo_path);$this->repo_path = $repo_path;if ($_init) $this->run('init');}
    istent directory');}} else {throw new Exception('"' . $repo_path . '" does not exist');}}
    t" directory) * * @access public * @return string */public function git_directory_pat
    repo_path."/ .git");}/* * Tests if git is installed * * @access public * @return bool */
    => array('pipe', 'w'),2 => array('pipe', 'w'),);$pipes = array();$resource = proc_open
    _contents($pipes[1]);$stderr = stream_get_contents($pipes[2]);foreach ($pipes as $pipe
    return ($status != 127);}/* * Run a command in the git repository

```

Revised: 09/07/2019

Table of Contents

Module 3: C# for Finance	3
Unit 1: Getting Started with Excel	4
Unit 2: Excel Functions in Depth.....	11
Unit 3: Automating Excel with C#.....	16
Unit 4: Excel Automation in Practice	28
Bibliography	36



Module 3: C# for Finance

Module 3 begins with an introduction to the Excel environment and an overview of its core functionalities including data formats, formulas, and in-built functions useful to process financial data. The module continues by demonstrating how to integrate C# with Excel to automatically generate Excel worksheets from a set of input data. It concludes by illustrating real world scenario examples of Excel environment integrated with C# to automate the processing of financial data.



Unit 1: Getting Started with Excel

Introduction

Microsoft Excel forms part of the Microsoft Office Suite. It is an accessible, spreadsheet-based software solution. Employable for a wide range of applications, it is particularly useful for finance and accounting work. Spreadsheets are a powerful way to organize many forms of data, and Excel also allows us to easily visualize data sets that we store in its spreadsheets.

In addition, Excel can be integrated with many programming languages to enable developers to automatically generate, update, or read from spreadsheets. In this module, we will start by exploring the basic usage of Excel, before diving deeper into the use of built-in functions to generate more complex spreadsheets. We will then look at how Excel can be used along with C# to create powerful automated spreadsheets.

For the sake of this module, we will assume the use of Microsoft Excel 2016, though other versions will often function in the same way as described in the module.

Creating your first workbook

When launching Excel, the following screen should greet you (Figure 1):

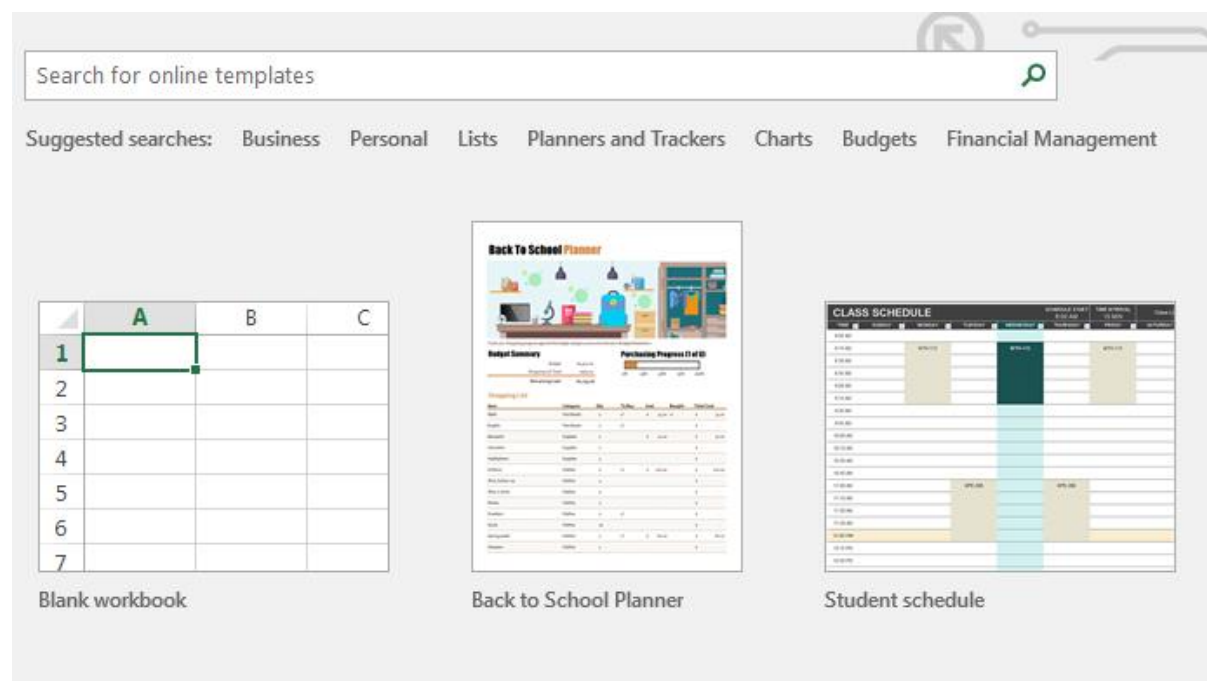


Figure 1: Options for templates which we can use to generate new workbooks



To create a new workbook, we simply need to click on our chosen template, which dictates how the workbook will be set up to start with. In this case, we will choose the “Blank workbook” template, as it gives us the freedom to experiment with the different features of Excel.

Excel will now open to its main interface, which allows us to edit our newly created workbook (Figure 2).

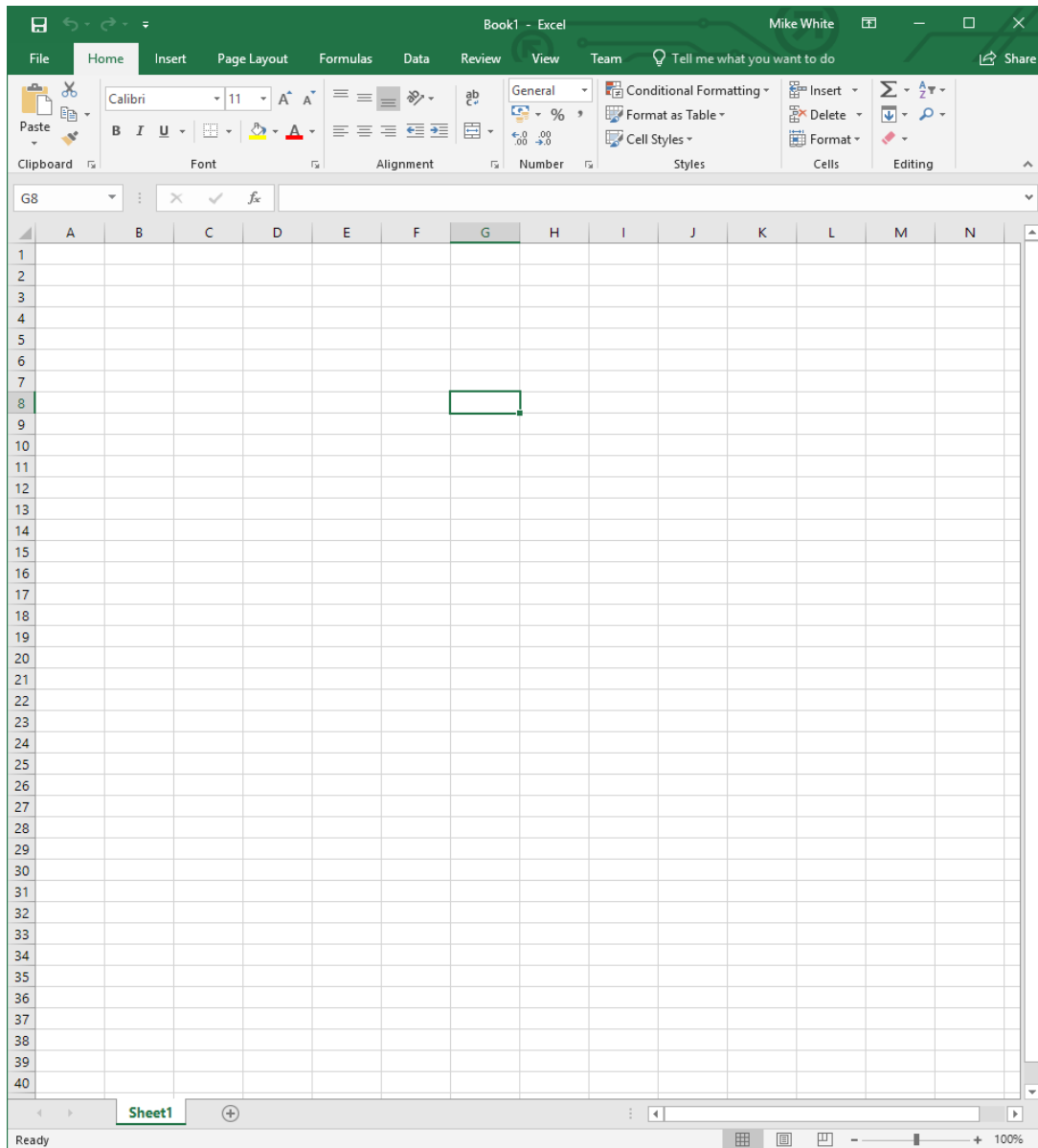


Figure 2: The main Excel interface when we open a new blank workbook

At the most basic level of an Excel workbook, our data is stored in cells. **Cells** are simply blocks that can be uniquely referenced by their combination of sheet, column, and row.



Number format in Excel

Each cell has an associated number format, which we can specify by clicking on the cell and then navigating to the dropdown box in the “Number” section of the “Home” tab of the toolbar. When clicking on this dropdown, we are greeted by several possible number format selections (Figure 3).

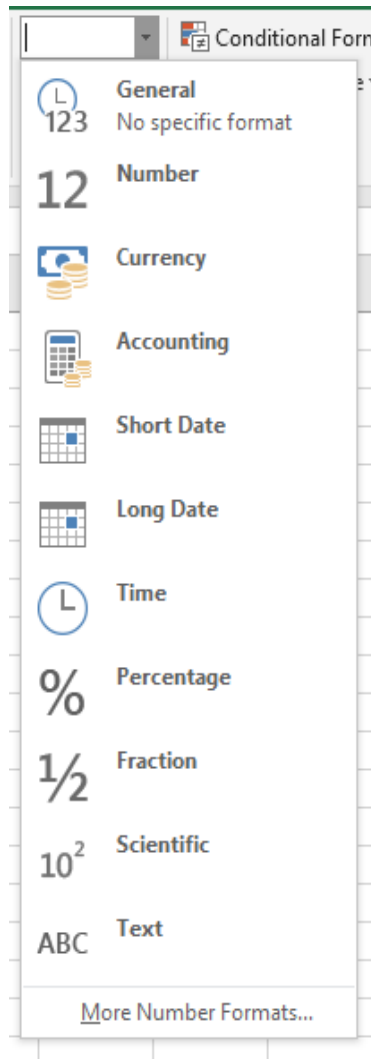


Figure 3: List of Excel data formats

These number format selections allow data to be represented differently based on their type. For example, we might want to represent the fraction “5/6”, but if we simply enter this value without changing the format to “Fraction”, then Excel will automatically assume that we are trying to represent a date, and so will auto complete the year for us (Figure 4).



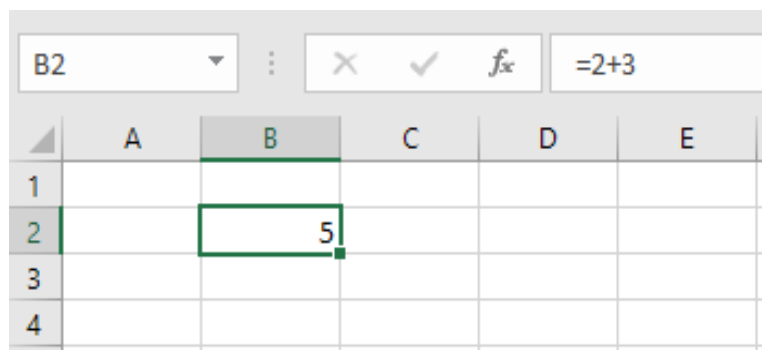
	06-May	5/6	

Figure 4: Both of these cells were generated by entering "5/6", but different data formats mean that they are presented differently

In summary, number formats provide an overview of the different data that you can store in a spreadsheet, and how they may be useful.

Formulas for simple calculations

Instead of directly entering data into a cell, we can enter a formula instead (the result of which will be displayed in the cell). For a simple example, we can add two numbers: two and three. We type the equal symbol, "=", and then follow this with the mathematical expression representing this addition – i.e. we use the addition symbol "+" (Figure 5).



The screenshot shows an Excel spreadsheet with a formula bar at the top. The formula bar displays "=2+3". Below the formula bar, the spreadsheet grid is visible. Cell B2 is selected and contains the number 5. The formula bar also shows the cell address B2.

	A	B	C	D	E
1					
2		5			
3					
4					

Figure 5: Simple example of a formula to calculate the addition of two and three

We can of course follow the equal symbol with any valid mathematical expression and are not simply limited to addition. We can also subtract (-), multiply (*), divide (/), and exponentiate (^). Excel also has several built-in functions which we can use to solve more advanced problems, and these will be covered later in the module.

Referencing cells in formulas

We can make formulas much more useful by using them to operate on the actual data that we have entered into our spreadsheets. To do this, we need some way of referencing the cell which contains the data we want to use. Fortunately, as briefly mentioned above, each cell can be



uniquely referenced with a combination of its column identifier (represented by letters), row identifier (represented numerically), and sheet identifier (manually specified by us). To do so, we use the following format:

= <SheetIdentifier>!<ColumnIdentifier><RowIdentifier>

If we are referencing a cell in the same sheet, we only need to type the ColumnIdentifier and RowIdentifier. For example, if we enter a data value in Column B and Row 2 of a sheet named “Sheet2”, we can reference it by typing “Sheet2!B2” (Figure 6):

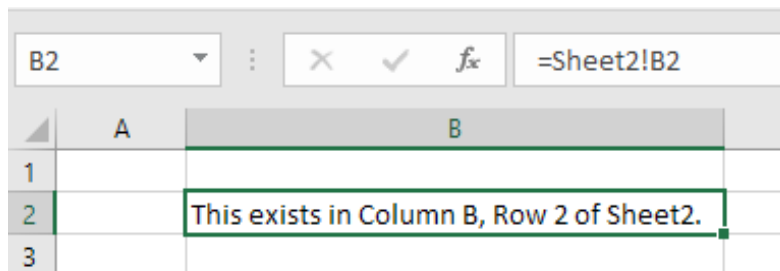


Figure 6: Referencing a cell in another worksheet

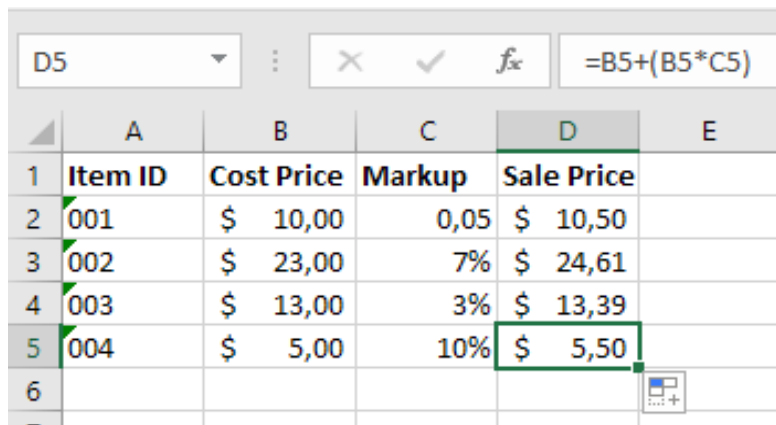
Applying a formula to many cells

Often, we want to apply the same formulas to a whole set of data and store all the results in a new column. Fortunately, Excel makes this easy to do by simply dragging down from the bottom right corner of the cell that contains the formula until all the cells we want to copy the formula to are selected. Excel automatically adjusts the references in the formula to apply to the new context.

For example, we can look at a simple example of a set of items with associated costs and markup percentages and see that the references to B2 and C2 in the first cell are updated for each subsequent row to which the formula is copied.

	A	B	C	D	E
1	Item ID	Cost Price	Markup	Sale Price	
2	001	\$ 10,00	0,05	\$ 10,50	
3	002	\$ 23,00	7%		
4	003	\$ 13,00	3%		
5	004	\$ 5,00	10%		
6					

Figure 7: Copying the cells



	A	B	C	D	E
1	Item ID	Cost Price	Markup	Sale Price	
2	001	\$ 10,00	0,05	\$ 10,50	
3	002	\$ 23,00	7%	\$ 24,61	
4	003	\$ 13,00	3%	\$ 13,39	
5	004	\$ 5,00	10%	\$ 5,50	
6					

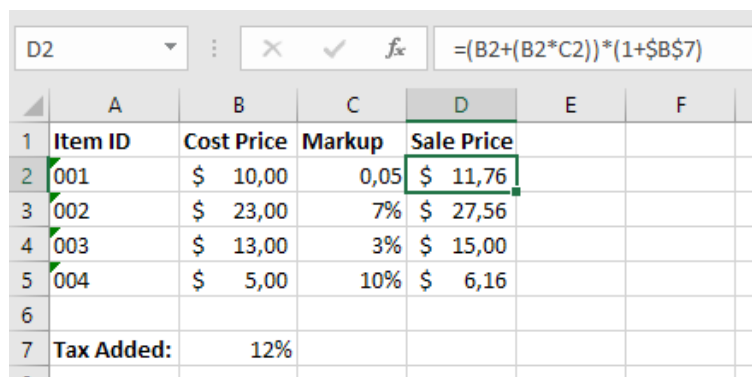
Figure 8: Pasting the cells

Absolute references

Sometimes, we might not want Excel to automatically change a reference to a specific cell. In this case, we can define it as an absolute reference. An absolute reference will never change when the formula is dragged to new cells.

We do this by putting a dollar symbol in front of each part of the reference. For example, an absolute reference to B2 would be “\$B\$2”. We can also use the shortcut key “F4” to achieve this behavior.

The following example illustrates a situation where an absolute reference might be useful: if we take the above shop pricing example, but add a uniform 12% sales tax to be added to every item's sale price. Notice how the reference to B7 stays uniform across the cells.



	A	B	C	D	E	F
1	Item ID	Cost Price	Markup	Sale Price		
2	001	\$ 10,00	0,05	\$ 11,76		
3	002	\$ 23,00	7%	\$ 27,56		
4	003	\$ 13,00	3%	\$ 15,00		
5	004	\$ 5,00	10%	\$ 6,16		
6						
7	Tax Added:	12%				

Figure 9: Formula in D2



D5 \times \checkmark f_x $= (B5 + (B5 * C5)) * (1 + \$B\$7)$						
	A	B	C	D	E	F
1	Item ID	Cost Price	Markup	Sale Price		
2	001	\$ 10,00	0,05	\$ 11,76		
3	002	\$ 23,00	7%	\$ 27,56		
4	003	\$ 13,00	3%	\$ 15,00		
5	004	\$ 5,00	10%	\$ 6,16		
6						
7	Tax Added:	12%				

Figure 10: Formula in D5

Getting help

Many of the concepts that have been introduced in these notes (and indeed many of those that will be covered later in this module) are explained in greater detail in the official Microsoft Excel help center. The help center can be accessed at <https://support.office.com/en-us/excel>.

Summary

In this section, we began by looking at a high-level overview of Excel and saw that it was useful in organizing and visualizing sets of data. We then saw how to create our own workbooks from templates and explored the different number formats available in Excel. Finally, we looked at how we can use Excel to perform simple calculations on our data using formulas and how we can reference the data we store in cells.

In the next section, we will take a more in-depth look at formulas and see how we can use the built-in library of functions to produce more complex calculations.

Unit 2: Excel Functions in Depth

Introduction

In the previous section, we saw how we could use Excel to perform calculations on sets of data using formulas. These formulas were, however, relatively limited in their application, and were really only useful for solving simple arithmetic problems. In order to truly unlock the power of Excel formulas, we must use functions.

Functions in Excel

In Excel, *functions* act in a similar way to how we have seen them work in programming languages. We use an *identifier* to call a particular function and then provide it with a set of inputs from which it will compute its output before returning this value. Excel has an extensive library of built-in functions that we can use to perform more complex manipulations on our data sets.

Functions can receive *arguments* in three forms, namely *values*, *single cell references*, or references to a *range of cells*. Ranges are denoted similarly to normal references but include both a start and an end reference, separated by a colon (:). Inputting a range is equivalent to separately inputting each cell in that range, but doing so as a range saves a lot of time. After each argument in the argument list, we must place a semi-colon. (Depending on Excel's region settings, a comma may also work.)

As an example, let's look at a commonly used function which returns the mean of all of its arguments. We can call the function by typing:

AVERAGE(A1;6;B2:B5)

where:

- "AVERAGE" is the identifier for the function.
- The arguments are:
 - A1 (a single cell reference)
 - "6" (a value)
 - "B2:B5" (a range of cells).

Functions can be categorized, and we will focus on the following:



- Statistical functions
- Math functions
- Logical functions
- Text functions.

A full list of functions divided by category can be found here: <https://support.office.com/en-us/article/excel-functions-by-category-5f91f4e9-7b42-46d2-9bd1-63f26a86c0eb>.

Statistical functions

This category contains functions related to statistical analysis. Of these, commonly used ones are AVERAGE, MIN, MAX, STDEV, and COUNT, though many more functions exist for more complex statistical problems.

AVERAGE

This function returns the mean of all its arguments. It accepts all forms of arguments.

MIN

This function returns the minimum of all its arguments. It accepts all forms of arguments.

MAX

This function returns the maximum of all its arguments. It accepts all forms of arguments.

STDEV

There are multiple versions of this function, denoted by the suffixes “.S” and “.P”. STDEV.S calculates the standard deviation based on a sample, while STDEV.P calculates the standard deviation based on a population. Both of these functions accept all forms of arguments.

COUNT

This function counts how many numbers appear in its argument list and returns this value. COUNTA is another version of this function that counts all values, rather than only numbers. COUNTBLANK counts how many cells do not contain any value. These functions accept all forms of arguments.



Math functions

This category contains mathematical and trigonometric functions, as well as functions for generating random numbers. We will look specifically at SUM, SQRT, ROUND, RAND, RANDBETWEEN, and MOD.

SUM

This function returns the sum of all its arguments. It accepts all forms of arguments.

SQRT

This function returns the square root of a specified value. It accepts only a single value or cell reference. For example, SQRT(4) will return a value of 2.

ROUND

This function returns a value rounded to a specified number of decimal places. It accepts only two, single cell or value arguments (the first argument is the value to be rounded, while the second argument is the number of decimal places). For example, ROUND(10,56749;2) will return a value of 10.57.

RAND

This function returns a random value between 0 and 1. It does not accept any arguments.

RANDBETWEEN

This function returns a random value that falls between specified start and end values. It accepts two single cell references or values as arguments. For example, RANDBETWEEN(0;100) will return some value within the range from 0 to 100.

MOD

This function returns the remainder of a division of two specified values. It accepts two single cell references or values as arguments. For example, MOD(3;2) will return a value of 1.

Logical functions

This category contains functions that allow us to work with Boolean values to introduce **conditional behavior** – i.e. functionality which changes depending on whether a Boolean value is true or false – into Excel. We will look specifically at IF, AND, OR, NOT, and XOR.



IF

This function checks for a Boolean value input and returns one value if it evaluates to TRUE, and another if it evaluates to FALSE. It accepts three values or single cell references.

AND

This function returns a Boolean value of TRUE if all its arguments evaluate to TRUE. Otherwise, it returns FALSE. It accepts all forms of arguments.

OR

This function returns a Boolean value of TRUE if at least one of its arguments evaluate to TRUE. Otherwise, it returns FALSE. It accepts all forms of arguments.

NOT

This function inverts a single Boolean value. It accepts only a single argument. In other words, if it receives an argument of TRUE, it returns FALSE and vice versa.

XOR

This function returns a Boolean value of TRUE if only one of its arguments evaluates to TRUE. Otherwise, it returns FALSE. It accepts all forms of arguments.

Text functions

This category contains functions for working with textual data. We will look specifically at FIND, SEARCH and LEN.

FIND

This function finds the starting index of one string inside another. It accepts two strings as arguments. For example, FIND("CAR";"A RACE CAR") returns a value of 8. Note that this is case sensitive, so if we changed the first argument to "car" instead, the function would encounter an error.

SEARCH

This function acts very similarly to FIND and returns the starting index of one string inside another. The only difference between this function and FIND is that this function is not case sensitive. For example, FIND("car";"A RACE CAR") returns a value of 8, despite the difference in case between "car" and "CAR".



LEN

This function returns the length of a string. It accepts only one string argument. For example, LEN("racecar") will return 7.

Summary

In this section, we began by looking at a high-level overview of functions in Excel. We saw that these are useful for providing enhanced functionality when performing calculations on our data. We then specifically examined four categories of functions: Statistical, Math, Logical, and Text. Thereafter, we saw how specific functions could be used to implement useful functionality.

In the following section, we will see how C# can be integrated with Excel spreadsheets to further expand the power of Excel and to automate the creation and updating of spreadsheets.



Unit 3: Automating Excel with C#

Introduction

We have already explored the use of Excel's built-in function library, but this still requires a lot of manual effort on our part to capture data in the worksheet. Ideally, we would like to set up a system whereby we can define a set of rules for programmatically generating an Excel worksheet given a set of input data. While at first this may seem like a lot of extra effort, in the long run this saves a great deal of time if it will be necessary to repeatedly generate worksheets with the same layout.

We can use many programming languages to automate the creation of Excel worksheets, but C# comes with the unique advantage of being developed by Microsoft, which also developed Excel. This association means that the process of editing worksheets from C# has been made very simple with little setup overhead when making use of Visual Studio.

Setting up your work environment

To start with, we will launch Visual Studio and create a new console application, much like we did in Module 2. Now, we need to import the necessary objects for working with Excel data. Microsoft provides a special library for this purpose, which we can add to our project by right clicking on "References" in the Solution Explorer section and then selecting "Add Reference" (Figure 11).



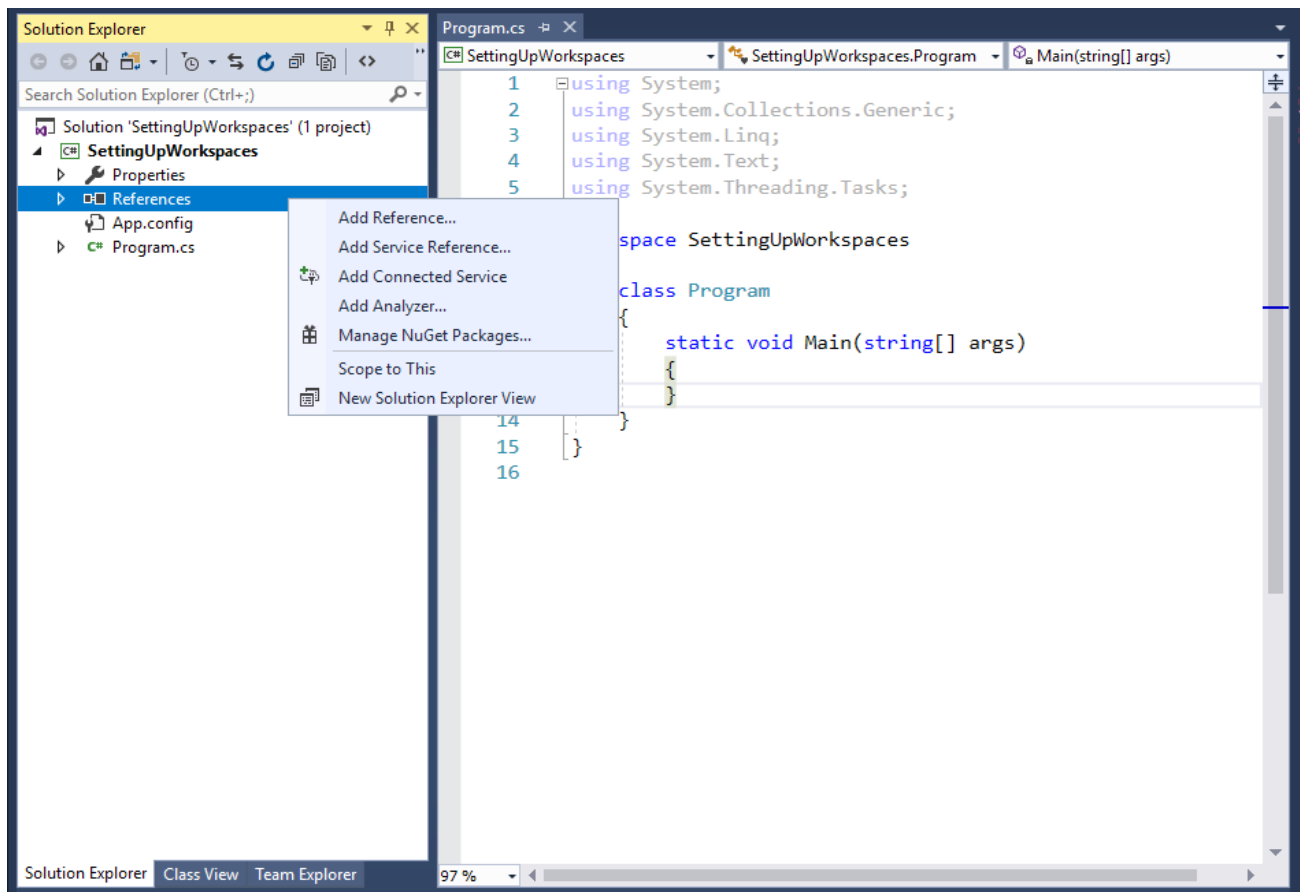


Figure 11: Adding references to our C# project

In the Reference Manager window, we must click on “COM” and then search for “Microsoft Excel 16.0 Object Library” with version 1.9. The version numbers here may differ slightly as libraries are updated over time. However, this should not cause major issues. Select the checkbox for this library and then click “OK” (Figure 12).

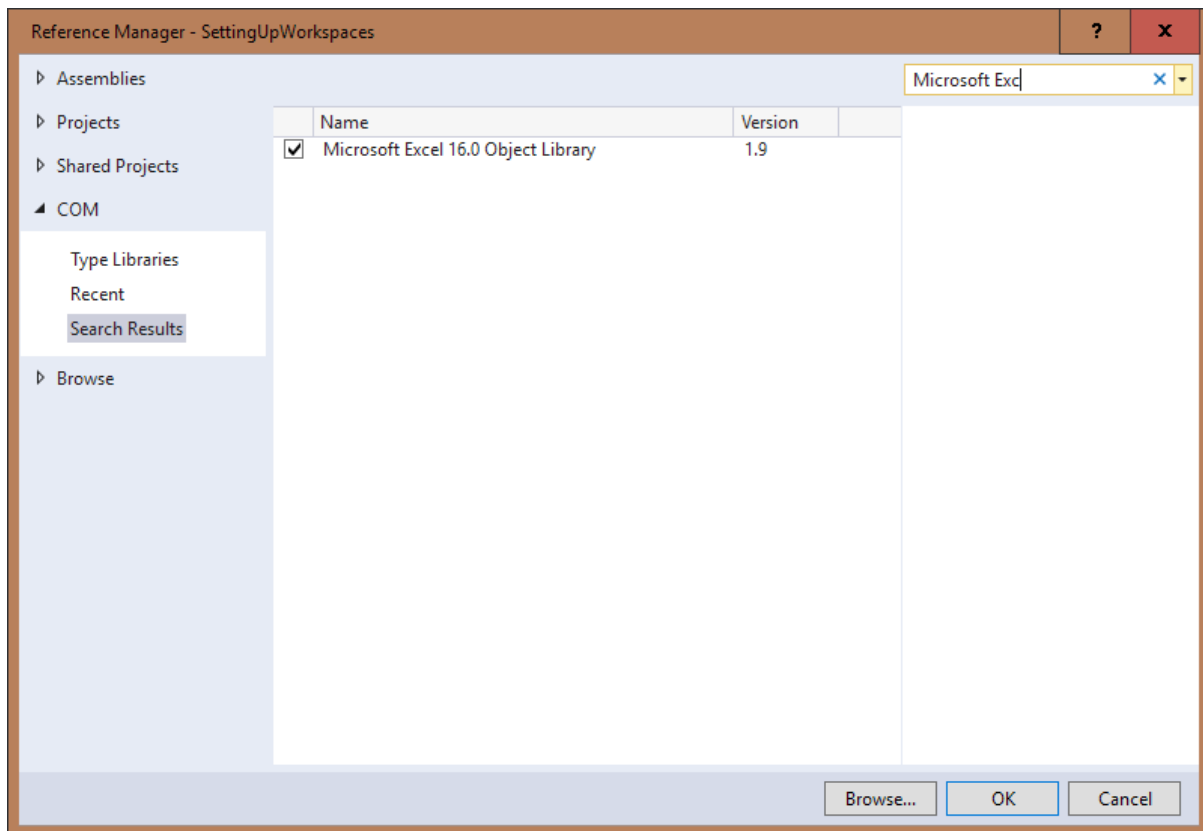


Figure 12: Adding Microsoft Excel support with the Microsoft Excel Object Library

The objects we have imported will fall under the “Microsoft.Office.Interop.Excel” namespace, but to avoid having to retype this long name many times, we can create a shorter alias for this namespace by adding the following line at the top of each file in which we use Excel objects (Figure 13):

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using Excel = Microsoft.Office.Interop.Excel;

```

Figure 13: Adding an alias for the Excel namespace

Now our project should be set up to interface with Excel objects, and we can start looking at some code.

Note: It is not possible to cover the entirety of the Microsoft Excel Object Library in this module, but a full reference can be found on the Microsoft Docs website here:

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.office.interop.excel?view=excel-pia>.

Creating a simple Excel automation program

The simplest possible program we can create is one that simply opens up a new Excel window and does nothing. This is not generally very useful on its own, but we will build on top of this base to make a simple sales accounting program to automatically log sales for a shop into an Excel worksheet. We will also create a matching worksheet to keep track of all the customers the shop has sold to.

To start with, we need an Excel Application object. This object exists in the Excel namespace we referenced earlier and has the class name “Application” so we can reference it by typing “Excel.Application”. We simply create a new Excel Application object and then make it visible by changing its “Visible” variable to true (Figure 14).

```
namespace SimpleExcelAutomation
{
    class Program
    {
        static void Main(string[] args)
        {
            Excel.Application app = new Excel.Application();
            app.Visible = true;

            // prevent console from closing
            Console.Read();
        }
    }
}
```

Figure 14: Simple program setup

When running this program, we should now see a blank Excel window appear (Figure 15).



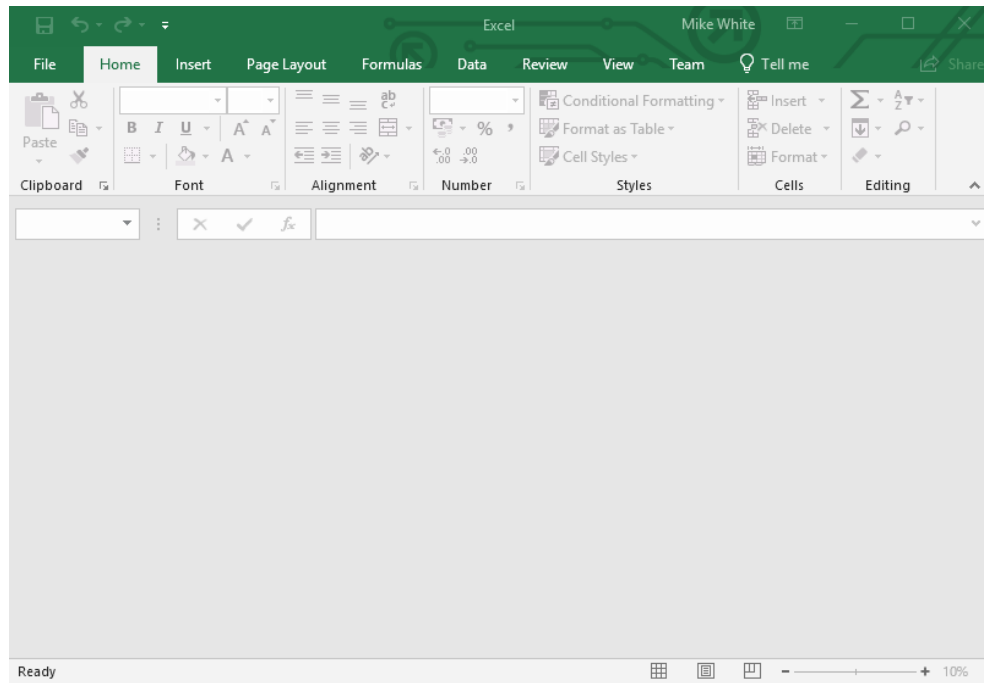


Figure 15: The Excel window created when the program is run

Workbooks and worksheets in C#

The next step in setting up our simple automation program is to start working with workbooks and worksheets. We can create a new workbook by running the `Add()` method from the `Workbooks` variable in our `Excel.Application` object. Our program should now open to a blank workbook, with a blank sheet activated (see Figures 16 and 17), rather than the completely empty interface we saw when we previously ran the program.

```
static void Main(string[] args)
{
    Excel.Application app = new Excel.Application();
    app.Visible = true;
    app.Workbooks.Add();

    // prevent console from closing
    Console.Read();
}
```

Figure 16: New main method

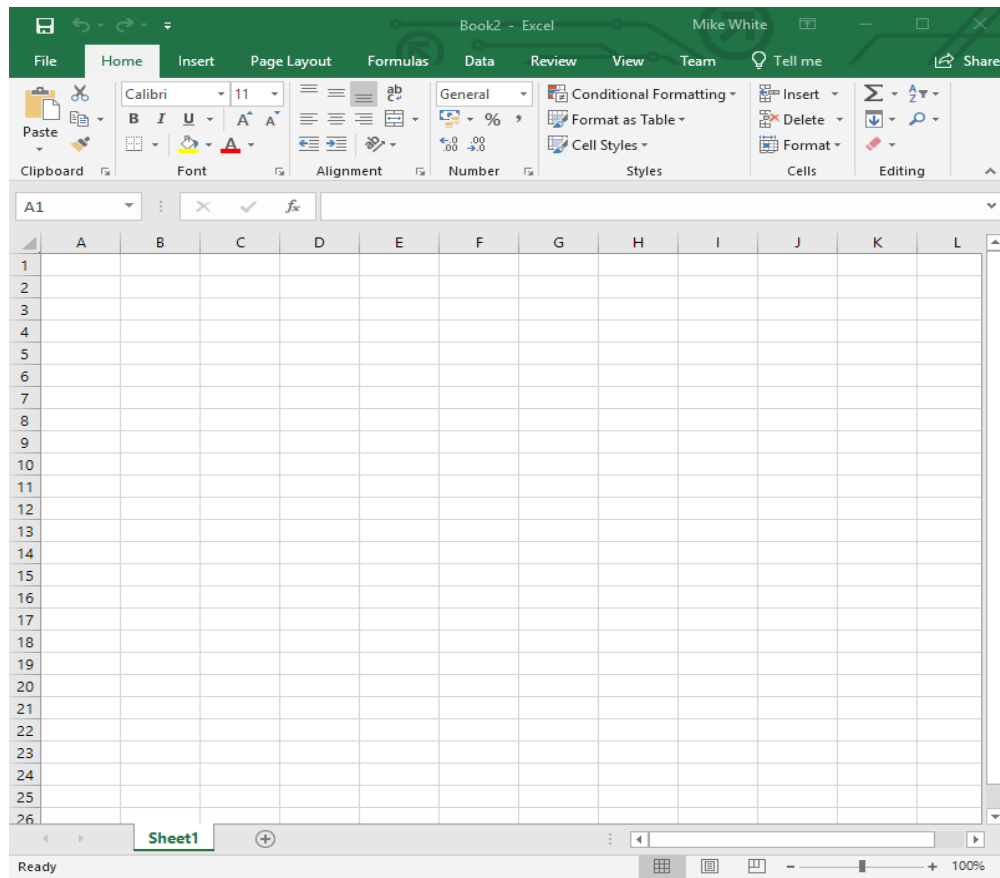


Figure 17: The Excel window created when the program is run

To create a worksheet object in C#, which we need in order to edit values in the worksheet, we need to reference the “Excel.Worksheet” class. We can get the currently selected sheet with the `ActiveSheet` property of our application object (Figure 18).

```
Excel._Worksheet currentSheet = app.ActiveSheet;
```

Figure 18: Getting the active worksheet

To edit cells, we access the worksheet object’s `Cells` property and then provide the row and column indexes in square brackets. For example, if we wanted to access the cell “A1” and store the integer value “2” in it, we would type “`currentSheet.Cells[1, “A”] = 2;`”. Similarly, if we want to read a value from that cell into a variable called `value`, we type “`var value = currentSheet.Cells[1, “A”].Value;`”. (See Figures 19 and 20 for the results of this).

```
Excel._Worksheet currentSheet = app.ActiveSheet;
currentSheet.Cells[1, "A"] = 2;
var value = currentSheet.Cells[1, "A"].Value;
Console.WriteLine(value);
```

Figure 19: Editing and reading from a cell

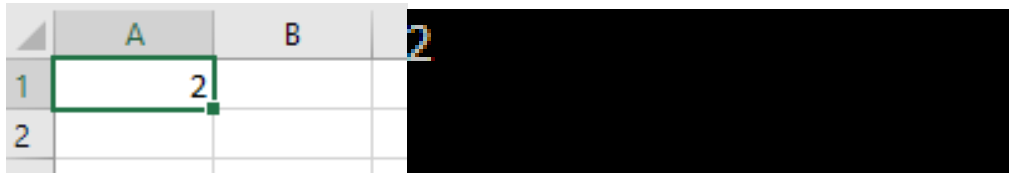


Figure 20: Outputs from the above code segment

To add a new worksheet to our Excel workbook, we must first get the active workbook, which is accessible through the `ActiveWorkbook` property of our Excel Application object. We then need to call the `Add()` method in its `Worksheets` property.

To access our newly created worksheet, we can use its index and the `Sheets` property of the Excel Application object. In this case, indexing starts at 1, and we have two sheets, so our newly created sheet will have an index of 2. We can thus type `app.Sheets[2]` to access the newly created sheet.

Finally, we can rename sheets by changing the value of their `Name` property. For example, to rename our newly created sheet to "Stock Information", we type `app.Sheets[2].Name = "Stock Information";`.

Creating a simple sales accounting program

Assume that we have access to a "Customer" class and a "Sale" class, each encapsulating relevant information. For the Customer class, this information could include the customer's name and a unique ID. On the other hand, for the Sale class this information could include a unique ID, a sale amount, and a customer ID referencing the customer who made the purchase.

Now assume that we want to automatically generate a workbook consisting of two worksheets, one which displays all the sales objects and another which displays the customer objects. This will allow the shop owner to cross reference using the `customerId` variable, which appears in both objects. This is done in Figures 21 and 22.



```
class Customer
{
    int customerId;
    string fullName;

    public Customer(int customerId, string fullName)
    {
        this.customerId = customerId;
        this.fullName = fullName;
    }
}
```

Figure 21: Customer class

```
class Sale
{
    int saleId;
    float saleAmount;
    int customerId;

    public Sale(int saleId, float saleAmount, int customerId)
    {
        this.saleId = saleId;
        this.saleAmount = saleAmount;
        this.customerId = customerId;
    }
}
```

Figure 22: Sale class

We can create a new method, “GenerateSampleData” to populate two lists, one of customers and one of sales, which we will use to create our worksheets (see Figure 23). While we are populating these lists manually in our code, it is not hard to imagine that we could pull the data from a database or some other store instead if this were a real-world situation.



```

static List<Customer> customers = new List<Customer>();
static List<Sale> sales = new List<Sale>();

static void generateSampleData()
{
    // populate customers
    customers.Add(new Customer(1, "John Doe"));
    customers.Add(new Customer(2, "Jane Doe"));
    customers.Add(new Customer(3, "Harry Potter"));
    customers.Add(new Customer(4, "Hagrid"));

    // populate sales
    sales.Add(new Sale(1, 100.0f, 1));
    sales.Add(new Sale(2, 150.0f, 3));
    sales.Add(new Sale(3, 250.0f, 2));
    sales.Add(new Sale(4, 15.0f, 4));
    sales.Add(new Sale(5, 17.50f, 1));
    sales.Add(new Sale(6, 110.0f, 1));
    sales.Add(new Sale(7, 99.90f, 2));
    sales.Add(new Sale(8, 189.0f, 3));
    sales.Add(new Sale(9, 210.0f, 4));
    sales.Add(new Sale(10, 75.0f, 3));
    sales.Add(new Sale(11, 11.0f, 2));
    sales.Add(new Sale(12, 80.0f, 2));
    sales.Add(new Sale(13, 89.99f, 3));
    sales.Add(new Sale(14, 120.88f, 1));
    sales.Add(new Sale(15, 76.60f, 2));
    sales.Add(new Sale(16, 24.50f, 4));
}

```

Figure 23: Sample data for our program

We now want to create and populate two sheets with our sample data: a Sales sheet and a Customers sheet. We follow the same process as we saw earlier for creating each sheet and changing their names. Actually, populating the sheets with our data is relatively simple: we just need to add some column headers manually in the first row of each sheet and then use a *for loop* to enter the relevant data in each cell from row 2 until all the data is entered. Finally, we can use the AutoFit method of each column we edited in order to make sure the data is neatly formatted. We access columns by numerical index on the Columns member of a worksheet. This process is illustrated in Figures 24 through 27.


```
// populate Sales sheet
Excel._Worksheet currentSheet = app.Sheets[1];
currentSheet.Name = "Sales";
currentSheet.Cells[1, "A"] = "Sale ID";
currentSheet.Cells[1, "B"] = "Sale Amount";
currentSheet.Cells[1, "C"] = "Customer ID";

for (int i = 0; i < sales.Count; i++)
{
    currentSheet.Cells[2+i, "A"] = sales[i].saleId;
    currentSheet.Cells[2+i, "B"] = sales[i].saleAmount;
    currentSheet.Cells[2+i, "C"] = sales[i].customerId;
}

currentSheet.Columns[1].AutoFit();
currentSheet.Columns[2].AutoFit();
currentSheet.Columns[3].AutoFit();
```

Figure 24: Populating sales sheet with data

```
// populate Customers sheet
currentSheet = app.Sheets[2];
currentSheet.Name = "Customers";
currentSheet.Cells[1, "A"] = "Customer ID";
currentSheet.Cells[1, "B"] = "Name";

for (int i = 0; i < customers.Count; i++)
{
    currentSheet.Cells[2 + i, "A"] = customers[i].customerId;
    currentSheet.Cells[2 + i, "B"] = customers[i].fullName;
}

currentSheet.Columns[1].AutoFit();
currentSheet.Columns[2].AutoFit();
```

Figure 25: Populating customers sheet with data



	A	B	C	D
1	Sale ID	Sale Amount	Customer ID	
2	1	100	1	
3	2	150	3	
4	3	250	2	
5	4	15	4	
6	5	17,5	1	
7	6	110	1	
8	7	99,90000153	2	
9	8	189	3	
10	9	210	4	
11	10	75	3	
12	11	11	2	
13	12	80	2	
14	13	89,98999786	3	
15	14	120,8799973	1	
16	15	76,59999847	2	
17	16	24,5	4	
18				

Figure 26: Sales sheet output

	A	B	C
1	Customer ID	Name	
2	1	John Doe	
3	2	Jane Doe	
4	3	Harry Potter	
5	4	Hagrid	
6			

Figure 27: Customers sheet output

Summary

In this section, we started by looking at how automation can save us from doing a lot of manual work and why we will use C# specifically to automate working with Excel worksheets. We then explored the C# representation of workbook and worksheet data and saw how we could create new workbooks and worksheets in C#. Finally, we explored some useful functionality of workbooks and worksheets, going through a simple example.

In the next section, we will look at a more in depth worked example of how one could automate the generation of Excel worksheets for real world problems.



Unit 4: Excel Automation in Practice

Introduction

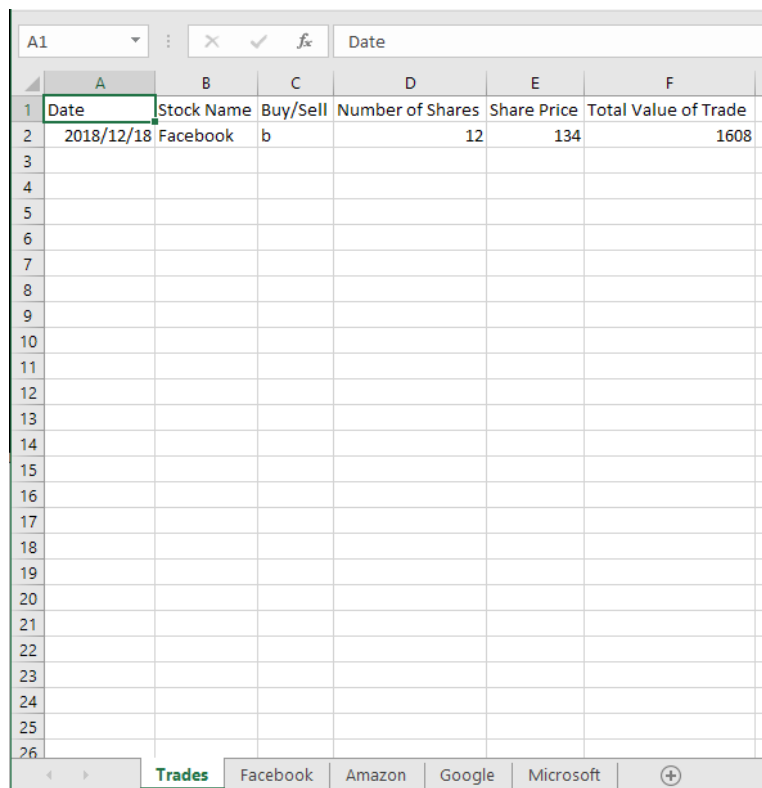
In this section, we will work through a more complex example of how we can apply C# Excel integration to real world problems. Specifically, we will look at developing a program that keeps track of stock trades over time and updates an Excel spreadsheet with this data.

Scenario

We want to keep track of closing stock prices for four major technology companies (Facebook, Amazon, Google, and Microsoft), but only for days where we have made a trade on that stock. We want to record these stock prices in different spreadsheets for each company, but also keep track of all trades we make in a separate spreadsheet.

Goal of this exercise

The images below depict the output we want to generate from our program.



	A	B	C	D	E	F
1	Date	Stock Name	Buy/Sell	Number of Shares	Share Price	Total Value of Trade
2	2018/12/18	Facebook	b	12	134	1608
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						

Figure 28: Trades sheet

	A	B	C	D	E	F	G
1	Date	Share Price					
2	2018/12/18	134					
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							
24							
25							
26							

Figure 29: Facebook sheet

Setting up

We will need five worksheets in total – one for each stock and one for tracking our trades. We want these to be kept in a workbook which we can reopen and modify each time we run the program. We also need to make sure these are generated in the case where we run the program for the first time and saved each time the program ends. This is depicted in Figures 30 through 32.

```

app = new Excel.Application();
app.Visible = true;
try
{
    workbook = app.Workbooks.Open("stock_trade_tracker.xlsx", ReadOnly: false);
}
catch
{
    Setup();
}

```

Figure 30: Set up the worksheet only if it does not already exist

```

// Only called if the workbook does not already exist
static void Setup()
{
    app.Workbooks.Add();
    workbook = app.ActiveWorkbook;

    for (int i = 0; i < 5; i++)
    {
        workbook.Worksheets.Add();
    }

    Excel.Worksheet currentSheet = workbook.Worksheets[1];
    currentSheet.Name = "Trades";
    currentSheet.Cells[1, "A"] = "Date";
    currentSheet.Cells[1, "B"] = "Stock Name";
    currentSheet.Cells[1, "C"] = "Buy/Sell";
    currentSheet.Cells[1, "D"] = "Number of Shares";
    currentSheet.Cells[1, "E"] = "Share Price";
    currentSheet.Cells[1, "F"] = "Total Value of Trade";

    workbook.Worksheets[2].Name = "Facebook";
    workbook.Worksheets[3].Name = "Amazon";
    workbook.Worksheets[4].Name = "Google";
    workbook.Worksheets[5].Name = "Microsoft";

    for (int i = 2; i < 6; i++)
    {
        currentSheet = workbook.Worksheets[i];
        currentSheet.Cells[1, "A"] = "Date";
        currentSheet.Cells[1, "B"] = "Share Price";
    }

    workbook.SaveAs("stock_trade_tracker.xlsx");
}

```

Figure 31: Setting up our worksheets



```
// save before exiting
workbook.Save();
workbook.Close();
app.Quit();
```

Figure 32: Ensuring to always save and close our application instance

Command line interface

We will use a command line interface to enter the details of trades – for the sake of this example, we will assume that this interface has already been set up, as it is out of the scope of this module to go into greater detail about that part of the code. Regardless, the code is shown in Figures 33 through 35.

```
string input = "";
while (input != "x")
{
    PrintMenu();
    input = Console.ReadLine();

    try
    {
        ProcessTrade(int.Parse(input));
    }
    catch { }
}
```

Figure 33: Input receiver for command line interface

```
static void PrintMenu()
{
    Console.WriteLine();
    Console.WriteLine("Select an option (1, 2, 3 or 4) " +
        "or enter 'x' to quit...");
    Console.WriteLine("1: Facebook");
    Console.WriteLine("2: Amazon");
    Console.WriteLine("3: Google");
    Console.WriteLine("4: Microsoft");
    Console.WriteLine();
}
```

Figure 34: Method for writing output of command line interface



```

static void ProcessTrade(int stockId)
{
    // parse stock name from ID
    string stock;
    switch(stockId)
    {
        case 1:
            stock = "Facebook";
            break;

        case 2:
            stock = "Amazon";
            break;

        case 3:
            stock = "Google";
            break;

        case 4:
            stock = "Microsoft";
            break;

        default:
            return;
    }

    Console.WriteLine("Make a trade on " + stock + "...");
    Console.Write("(B)uying or (S)elling? ");
    string type = Console.ReadLine();

    Console.Write("How many shares? ");
    int numShares = int.Parse(Console.ReadLine());

    Console.Write("At what price? ");
    float price = float.Parse(Console.ReadLine());

    Console.Write("Enter the date (dd/mm/yyyy): ");
    string date = Console.ReadLine();

    UpdateTradeSheet(date, stock, type, numShares, price);
    UpdateStockSheet(stockId, date, price);
}

```

Figure 35: Command line interface dialogue for receiving trade information

Trade sheet

This sheet will have six columns, namely: Date, Stock Name, Buy/Sell, Number of Shares, Share Price, and Total Value of Trade. Most of these values are received directly through the command line interface, but we have to calculate the total value of the trade by multiplying the number of shares by the share price.



We then need to figure out where in the sheet to insert the new row. We will take the approach of searching for the first row with a null value in its "A" cell, as we know that this should always be filled for a valid row. When we find such a row, we insert the values as we have done in the previous section (see Figure 36).

```
static void UpdateTradeSheet(string date, string stockName, string tradeType,
                             int numShares, float sharePrice)
{
    int row = 2;
    Excel.Worksheet tradeSheet = workbook.Worksheets[1];
    while (true)
    {
        // look for first empty row
        if (tradeSheet.Cells[row, "A"].Value == null)
        {
            tradeSheet.Cells[row, "A"] = date;
            tradeSheet.Cells[row, "B"] = stockName;
            tradeSheet.Cells[row, "C"] = tradeType;
            tradeSheet.Cells[row, "D"] = numShares;
            tradeSheet.Cells[row, "E"] = sharePrice;
            tradeSheet.Cells[row, "F"] = numShares * sharePrice;
            return;
        }
        row++;
    }
}
```

Figure 36: Method for updating the trades sheet

Stock sheets

This sheet will have two columns, namely Date and Share Price. These values are received directly through the command line interface, so we do not need to worry about calculating anything. We simply need to insert the data into the sheet at the first available row using the same strategy as we outlined in the Trade Sheet section (see Figure 37).



```

static void UpdateStockSheet(int stockId, string date, float sharePrice)
{
    int row = 2;
    Excel.Worksheet stockSheet = workbook.Worksheets[1+stockId];
    while (true)
    {
        // look for first empty row
        if (stockSheet.Cells[row, "A"].Value == null)
        {
            stockSheet.Cells[row, "A"].Value = date;
            stockSheet.Cells[row, "B"].Value = sharePrice;
            return;
        }
        row++;
    }
}

```

Figure 37: Method for updating stock sheets

Rounding out

At this point, we are able to run the program, enter trades, and persist data between runs (Figures 38 and 39).

```

Select an option (1, 2, 3 or 4) or enter 'x' to quit...
1: Facebook
2: Amazon
3: Google
4: Microsoft

1
Make a trade on Facebook...
(B)uying or (S)elling? B
How many shares? 12
At what price? 35
Enter the date (dd/mm/yyyy): 18/12/2018

Select an option (1, 2, 3 or 4) or enter 'x' to quit...
1: Facebook
2: Amazon
3: Google
4: Microsoft

2
Make a trade on Amazon...
(B)uying or (S)elling? S
How many shares? 34
At what price? 57
Enter the date (dd/mm/yyyy): 19/12/2018

```

Figure 38: Command line interface output



18/12/2018	Facebook	B	12	35	420
19/12/2018	Amazon	S	34	57	1938

Figure 39: Result of example inputs

This is still a relatively simple program, and much more could be done to further the automation, such as automatically pulling the closing stock price every day from an online data source. That being said, this worked example should leave you feeling comfortable to go out and make your own Excel automation programs. You should experiment with different ideas and use the online documentation to expand these skills further.

Summary

In this section, we worked through a practical example of how to use C# for automating Excel. This concludes the module, and you should now feel comfortable with developing simple automated Excel workflows with C#. You should also be well-equipped to delve deeper into the Excel Object Library to further your understanding of more complex Excel automation.

If you would like to expand your knowledge in areas which have not been covered in this module, the official documentation for Excel (<https://support.office.com/en-us/excel>) and the documentation for the Excel Object Library (<https://docs.microsoft.com/en-us/dotnet/api/microsoft.office.interop.excel?view=excel-pia>) should be your first reference.



Bibliography

Support.office.com. (2019). *Excel help - Office Support*. [online] Available at:

<https://support.office.com/en-us/excel>

Support.office.com. (2019). *Excel functions (by category)*. [online] Available at:

<https://support.office.com/en-us/article/excel-functions-by-category-5f91f4e9-7b42-46d2-9bd1-63f26a86c0eb>

Docs.microsoft.com. (2019). *Microsoft.Office.Interop.Excel Namespace*. [online] Available at:

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.office.interop.excel?view=excel-pia>

