The Application object

introduction to the Application

This chapter will focus on the Application object. There are a number of significant details of Excel and its interaction with VBA that are controlled form the Application. In particular, the Application is responsible for providing access to calculation related properties and user display preferences.

The major features of the Application object include:

- Controlling the calculation
- Controlling events and other visual effects
- Controlling the StatusBar and providing feedback during a macro

TODO: any other items for this list?

The Application can do a number of other things related to Excel settings which will not be covered here.

Controlling calculations

When you are creating macro workflows, there are a number of tools at your disposal to control calculations flow. Before describing those tools, it's worth stepping back and discussing why you might want to control the calculation flow. There are a couple of common reason:

- Performance. Your code will run faster if you control the calculation process. This mainly involves disabling automatic calculation at key points.
- Accuracy. For some types of calculations, you need to tightly control the calculation flow for accuracy. This is often the case if you are building a spreadsheet that does some form of recursion or self reference.
- Usability. There are some situations where you are interacting with calculations and need to prevent the normal behavior. The most common is when you add Workbook events like `Change`.
- Profiling. If you are building a code profiler (i.e. a tool that tracks execution time of your code) you must control calculations in order to get the tracking right.

We'll get back to the applications, but it's also worth hitting the high points on how you can control the calculation. THe main knobs:

- Disable application wide
- Disable for a Worksheet

- Manually calculate a Range, Worksheet, or Application

THe types of changes you will make are fairly tightly couple to the applications above. In general, for performances nad usability reasons, you will be disable calculations. For accuracy or profiling applications, you will manually walking the calculation through.

Disabling calculations

The most common approach to controlling calculations is to simply disable them. To "disable" the calculations, is really to set the CalculationMode to Manual. It does not actually disable calculations, but instead it prevents the automatic calculations updates from firing like normal. The spreadsheet still maintains its normal model of calculations; they just don't run. This is an incredibly common approach to speeding up the performance of VBA code. The performance boost results form the fact that when VBA code executes, it is very tightly coupled to the normal Excel operations that take place. When you use VBA to set a `.Value` equal to some new value, it is functionally equivlabet to manually entering the value. Behind the scenes, Excel will fire off the normal Change events and update the dependent cells. This can become a bottleneck because VBA is able to rapidly fire off `.Value` changes. So rapidly, that processing all of the associated stuff can become a limitation. It is more or less guaranteed that you will run into this issue once you start writing VBA code. It is so common, that you will likely memorize the fix:

TODO: check this code

```
Application.CalculationMode = xlManual
Application.ScreenUpdating = False
Application.EnabledEvents = False

Application.EnabledEvents = True
Application.ScreenUpdating = True
Application.CalculationMode = xlAutomatic
```

Why does this code make everything faster? Well, it disables the slowest steps of Excel keeping track of your spreadsheet: visual updates, calculations updates, and other events. Turning all of those off will dramatically remove the bottlenecks to your code. What's the downside? Well, all of that stuff exists for a reason and it's possible you need it to keep functioning for some VBA operations. The non-calculatojn options ar ecovered in subsequent chapters, so we'll focus on the calculation part now.

What happens when you disable calculations? This is the key concept to understand to make sure your spreadsheets do not break when you go looking for performances. So what changes?

- Dependent cells are not updated. The “chain” is processed to its end on every update. Note, updates are sent downstream. Not all cells are updated, unless your Workbook contains a VOLATILE function.
- Charts and other functional graphics do not update. Internally, they don’t change at all. It’s not just a matter of the visuals being hidden, they are not calculated.
- Less important items:
  - Conditional formatting will not update.

So why might those things matter? The biggest reason is that if your VBA code depends on the state of the spreadsheets, then you are likely depending on calculations at some point. This means that you need to split you rcode into segments where you are not worried about cell values and those where you are. An example:

You are building a tool to process data from a CSV file. You have been told that you should delete data that is in the 0th to 10th percentile of a cost column. Unforateunyl, the data needs to be preprocessed in order to create an accurate cost column. Your CSV file contains a mess of extra text and other issues when need to be removed. Your workflow then is:

1. Import the CSV data
2. Preprocess the cost column to clean up the mess
3. Remove the rows below the 10th percentile.

You do a quick test and have no problem importing the CSV data. You’ve gone ahead and worked out the preprocessing logic… only took a couple calls to `Split` and `Trim`. You also went ahead and added a new column to compute the `PERCENTILE` based on the now cleaned result. This is looking great on your 100 row test data set. Your set your application loose on the 90,000 “real” data and quickly find that it will not complete within 10 minutes. What’s going on here? THe most likely problem is that your new PERCENTILE column is being reculated every time a preprocessed data cell is being added back to the spreadsheet. Your processing code looks like:

```
For Each rngCell in rngData
    rngCell.Value = CleanUpThisMess(rngCell)
Next
```

If `rngData` contains 90,000 cells, then your update code will call for at least 90,000 full Worksheet recalculations. Even worse, your PERCENTILE formula requires the entire column of data and so all cells have to update every time. `90,000 x 90,000` quickly becomes a problem.

So, why is the PERCENTILE function updating after every change? Do we really care what the intermediate values are? No.

This is why you want to have control of the calculations. in this case, you know that the processing code is not affected by the value of the PERCENTILE column. We only need the static data available in order to complete the processing. The fix here is to turn calcuaitliosn to manual during the processing step so that you do not incur 90,000 extra recalculations.

Once the processing is done, what do we do with the calculation mode? Well, that depends on how we do the deletion. There are a couple of options:

- Turn on an `AutoFilter` and do a FILTER-DELETE to remove all the rows in one shot.
- Iterate through the rows, one by one, and remove those which are in the 10th or lower percentiles

Looks like either will work, but hwo does calculation mode affect things? Well, if you go with the latter option, you will find that your PERCENTILES will update after each deletion. This is not the behavior you intended. You somehow want to remember the PERCENTILE value before you started the deletions. The solution then is to control the calculation mode again. Here, we are controlling things for **accuracy**. Our deletion approach will not work if we allow cells to update as we go.

Pro tip: if you are deleting cells, you should pretty much never go a row, column, or cell at a time. Instead you should build a `Range` of cells to be deleted using `Union` and delete them in one shot using `Delete`. This approach is called a `UNION-DELETE` and avoids all of the issues described above. It's also the fatest approach since it does a single deletion.

Controlling events and visuals

THe previous section focused on controlling calculations, generally for the sake of performance. When seeking better performances, there are two other changes that are commonly made. They rely on disable the screen updating also disabling events. The former is a pretty tame change and is a no brainer if you want performance. There are very few downsides to disabling the screen. Disabling events can give a big boost in performances also, but there are a couple more risks involved. In addition to performance, there are other times where you need to disable events in order for your code to work.

The most common code for performances is repated here for clarity:

```
Application.CalculationMode = xlManual
Application.ScreenUpdating = False
Application.EnabledEvents = False

Application.EnabledEvents = True
Application.ScreenUpdating = True
```

```
7 Application.CalculationMode = xlAutomatic
```

What does that code do? Again, it forces calculation to manual mode, the screen to not update, and events to not fire.

ScreenUpdating

Screen updating is one of those things that seems fairly silly if you are coming from another programming environment. In the vast majority of other programming settings, it is very uncommon for all of your changes to produce an immediate visual result. ON the one hand, this is doccifult and on the other, it is awful for performance. Exel takes the opposite approach since it is a user focused GUI that offers a scripting environment for uaotmatiojn. The default in Excel is that all of your commands will trigger the normal render and refresh that would have occurred had you manually made the change. IN practice, this can often produce a very cool effect where the commuter is quite literally doing all of the work for the user. once you get over the appeal of this, you will quickly be left with the question of: how much ƒaster will my code be if I do not process all of these visual updates? The answer: much faster.

There are very few risks to disabling the screen. The biggest risk is that you forget to enabled it again (usually because of an error) and then your usre will get odd behavior. In a lot of case, Excel will actually enable the screen anyways so the actual risk here is minimal.

The only real reason to leave the screen active is in the case where your automation can ataxy be usefully reviewed by the user. In that case, you may consider leaving the scren active so that the user can “watch” what is happening. Some users like to see what the code is doing.

EnableEvents

The more aggressive option in this chapter is to disable events form firing. This has the ffect of inmprpving performances because your code will be able to skip a number of potentially lsow steps. The downside f this approach is hat sometimes you need events to fire in order to achieve a desired result. This is especially the case if you wrote the event handlers.

For events, there is one extra considering for when yu might disable them. If you possibly making changes to the Workbook or Worksheet in an event, you will likely need to disable events while you make your changes. THe reason for this is to prevent an endless loop of your event handler processing the change you jut made. This is only relevant for a handful of event types (Selection and Change are common) but it happens to be the case that this is a problem on the most commonly used event handlers.

TODO: add an example of a full event handler that disables event handling

Controlling the StatusBar

Did you know you can control the STatusBar? Did you know that the area at the bottom of you screen is called the Status Bar? This is an area that can be used to provide feedback to the user. It can be quite helpful for a long running calculation where you intentionally disable all of the normal feedback than the user receives (screen updates, events, etc.). If you have done that, be aware that you can still provider an updating message to the user.

```
1 Application.StatusBar = "Some message"
```

This functionality is best used when you have a measurable way of providing progress feedback. This is commonly one when you are looping through a large list of item sand processing each one in turn. Depending on how quickly you can process a single item, you may choose to update the StatusBar to provider the progress to the user. Biggest issue to be aware of is that you can overload the STatusBar and create a situation where Excel is slow processing all of your STatusBra updates. If this is your problem, you can usually remedy this with a quick modulo function to only update the status every 10th iteration or similar.

TODO: add the general purpose status tracking code