overview of events

chapter will focus on using events to interact with the user and also to drive more functional spreadsheets

the major events to focus on are the `Workbook` events, including:

- `SelectionChanged` which can be used to track when the user clicks on something (do this if clicking in this row)
- `Changed` which allows for watching cells and doing specific things if the change was somewhere specific
    - using the Intersect technique to determine if the change was in an area of interest
- disabling events while making changed during events
- Application.OnWait event to trigger something to take place at a given interval

other ways to interact with events include via class modules with the WithEvents designation. These can be used to associate an object with an event and then wire up the code separate from the original macro code. This section might be useful for charting events if I ever get that code put together

Other areas where events take place is via the Ribbon and also via different controls that can live on the sheet. It would be good to discuss these as well.

common events

When talking about events, there are a couple of high level details to touch on:

- Where the events occur? That is, which object owns the event and how do you hook into it?
- When the does event occur?
- What are you allowed or not allowed to do while responding to the event?

For a spreadsheets, the events tend to occur within the objects of interest: Worksheet, Workbook, others (TODO: is that right?).

THe most common events are associated with the Workbook and Worksheet. If you want to tie into those events, you can typically just add a new handler using the VBE. This process is actually fairly straightforward. The task becomes more difficult when you want to tie into an event but you are not certain which object will fire the event, or you want to track an event that takes place outside of your code.

THe main considering when working with event handling code is that you need to be sensitive to the fact that you can enter an endless loop if you accidentally trigger the same event as the one you are responding to. This is surprisingly easy to do if you are tied into the `Changed` or `Selection_Chnaged` events which trigger quite frequently.

callbacks

ONe important point here is that all events are handled via callbacks. That is, you will create a Sub with a specific name and a specific signature which VBA then uses when the event occurs. This callback is important because it includes the information that you will need to discern what happened with the event. Each type of event can include its own specific parameters and your code can respond to them accordingly. This is important because if a cell was Changed, you will want different information than when a Worksheet was activated. In general, VBA is good about providing you useful lparmaeters so that your events can properly determine what took place. Despite the good parameters, you will very likely need to include If/Then code to determine if your event needs further processing.

specific events

For actual event handling code, it makes most sense to take a look at the specific events that can occur and show some techniques for handling them.

Worksheet

THe Worksheet hsa a number of events which are commonly used. These include:

- Changed
- SelectionChanged
- Activate

These events roughly correspond to their name and are easy enough to handle. The idea with these is that you have a specific worksheet that you want to monitor for a specific event. In that case, you add the event using the VBE and then add the handling code.

The most common approaches for using these events is to track what the user is doing and then provide some additional functionality based on their actions. There are a number of reasons that you might want to respond to their input:

- Advanced usability where you allow the act of selecting a cell or cells to determine that some macro should run. You could imagine on a certain sheet that selecting a new cell may mean “please load more data about this row” and the VBA responds accordingly.
- Validation of user input. It is common to watch what the user is changing and then determine if that change is allowed or not based on specific rules.
- Starting a new action with some user input. I have previously used editing a cell to trigger a goal seek on that cell. This was quite nice because the VBA would undo my edit and then goal seek the

previous cell to a new value based on its formula. This provided a very slick means of trigger goal seek without having to collect further user input.

- Refreshing some display. It's possible you set calculations to manual and then force a recalculate each itme the Worksheet comes into focus.

For the Worksheet, the typical flow is that you will create events at the Worksheet level only if you know than you will only want the code for a single Worksheet (or you are willing to duplicate it across Worksheets). If you want to have the same code run for *all* worksheets, you should look at the Workbook events which provide better views of the entire Workbook.

For some examples here, I will show you how to do the goal seek business along with a separate event which watches user Selection and then processes the cells accordingly.

TODO: add the code and description for the Goal Seek event

TODO: add the code nad desc for some event which activates on Select

Workbooks

For Workbooks, you have a lot of the same events as for a Worksheet. These events take the same parameters (TODO: is that right?) but allow you to watch for that event across all Worksheets in a Workbook.

Depending on what you are watching for, this either makes perfect sense or is a real burden with false events that are not interesting. You will hav eot determine the proper scope for your events depending on what you need them to do. There are not fast rules here. The summary of possibly events then includes:

- Changed
- SelectionCHange
- Activated
- Opened
- BeforeSave
- Other Save events
- Closing

TODO: add some callback parameters above (and verify names)

TODO: review other events to see what may be useful

These events are similar to Worksheets except that they give you additional hooks that only make sense for a Workbook, specifically related to Saving, Opening, and Closing. There events can be quite useful if you want to do some amount of processing before the file is saved. One common approach I have taken is to delete extraneous data from a workflow spreadsheet to reduce the size and save time for a file. This can

be used to great effect if your processing spreadsheet is generally pretty lean without the data it processes. You could also use this to delete a Chart that is large and having a big impact on file size. Once the file is opened, you can then use VBA to recreate the chart.

IN some cases, these events are used for that type of example where it seems like a lot of work to save some amount of hassle. Oftentimes, this is the case. You can spend a lot of time with event code to make it do exactly what you rwant. Sometimes for a user focused spreadsheet, however, this is the level of detail than tis required to ensure that everything will work every time for everyone.

TODO: add an example and desc for a data removal VBA

Application

There are also a couple of events that exist at the Application level. These include:

- OnWait
- TODO: any others?

Application.OnWait can be used to trigger an event at some point in the future. This can then be used to trigger a block of code which runs at an interval by having the triggered code start a new event in the future. In this ways, you can use VBA to start a timer which executes every so often.

TODO: add the OnWait code for a timer

TODO: find another examples

common patterns

There are a number of patterns that are very common with Events. These patterns typically exist to avoid causing a problem or to avoid extra work where possible. Most VBA is not performance critical, but it is possible for an event to be called hundreds of times for a given chucnk of code. Since this is true, you can start to have an immediate impact on performance if your event handling code includes a number of unnecessary steps. As a side note, this is a good reminder that when trying to speed up code, you will nearly always do better to add `Application.EnableEvents = False` before your performance critical code; this assumes that your VBA does not rely on events firing to function properly.

Intersect

The first is the `Intersect` technique to determine if a Range that was affected by an event was a Range of interest. With this approach, you define a Range which includes your “interesting” cells. You then do a `If Not Intersect(rngEvent, rngTarget)Is Nothing` to see if the intersection of the callback Range and the desired Range overlap. If they overlap, yhen you typically execute some code. This allows

you to quickly filter out Ranges which have changed but are not relevant to Athena code you need to run.

TODO: add a code sample here

Application.EnableEvents = False

One of the biggest gotchas with Events is that you can quickly and accidentally create an endless loop of Event code running if your event handler is able to retirgger the original event. This is quite common if you are looking at the Selection and then change the selected cell. The same can happen if you are using an event to watch for a change and then you respond with additional changes. Both of these accidents are so common, that you should seriously consider always disabling events in your handler. It is quite rare that you will need an other event to fire following your own processing.

The main thing to remember here is that you really need to enable events again. Excel will not do this for you. You can create odd situations if you have an error in your code that goes unchecked. This situation can mean that events are disabled. For really sensitive, user focused code, you should add a proper error handler and enable events following that.

To handle this event, the code is quite simple:

```
1 Sub EventHandler()
```

```
    'disable events
    Application.EnableEvents = False

    '' do some stuff

    're-enable events
    Application.EnableEvents = True
End Sub
```

more advanced events

This section will focus on using events in more advanced settings. In particular, the focus here will be on using Class Modules to allow for events to be attached to arbitrary objects that are not necessarily known at compile time. This is an advanced approach that is typically not required. Where it may be helpful is if you are building library code that needs to work in a range of settings. It may also be needed if you are trying to attach events to a Worksheet that will not exist until some other VBA has been run. In this case, you will attach to the same events as above, but you will add the event after the Worksheet has been created. It's worth noting for that specific example etha the Workbook can handle a large number of events

on the Worksheet and will work for Worksheets that were created later.

TODO: add a section explaining how to use WithEvents

TODO: add some examples of attaching events to a new Worksheet.

SetUpKeyboardHooksForSelection.md

```
Public Sub SetUpKeyboardHooksForSelection()



    'SHIFT =    +
    'CTRL =     ^
    'ALT =      %

    'set up the keys for the selection mover
    Application.OnKey "^%{RIGHT}", "SelectionOffsetRight"
    Application.OnKey "^%{LEFT}", "SelectionOffsetLeft"
    Application.OnKey "^%{UP}", "SelectionOffsetUp"
    Application.OnKey "^%{DOWN}", "SelectionOffsetDown"

```

```
    'set up the keys for the indent level
    Application.OnKey "+^%{RIGHT}", "Formatting_IncreaseIndentLevel"
    Application.OnKey "+^%{LEFT}", "Formatting_DecreaseIndentLevel"

End Sub
```