

PennyPincher 1.0

Executive Summary

The purpose of my VBA project is to provide an automated tool that scans a list of items purchased from Staples, downloads the latest prices for the items in the inventory, and requests partial refunds if prices for any of the purchased items have dropped.

The resulting program, PennyPincher 1.0, has the potential to save small businesses hundreds or even thousands of dollars every year by ensuring that the business takes full advantage of Staples's 14 day price guarantee.

The user of this program enters key pieces of data from a Staples purchase into a spreadsheet by using a user form that is accessed via a button on the spreadsheet. The user may then choose one of two options to check for price changes.

One option is a onetime check that checks all the prices, sends out e-mails requesting partial refunds for items that have dropped in price, updates the data in the spreadsheet, then terminates. The second option sets the program to repeat the price check at regular intervals of the user's choosing.

One of the most valuable aspects of this program is that, when it does find an item that has a new, lower price, the program actual sends an e-mail to Staples customer service detailing the drop in price and requesting a refund. This ensures that no refund opportunity is ever missed due to delay or forgetfulness on the user's part.

Implementation

Step 1: Project Design

The first step in creating PennyPincher 1.0 was to design the project. This may seem like an obvious step, and it may seem like this step was accomplished once I had decided to write an automated price match program. However, there were several key decisions that had to be made about the scope of the project, and I had to do a lot of research to discover what key data points would be needed from the user for Penny Pincher to work correctly.

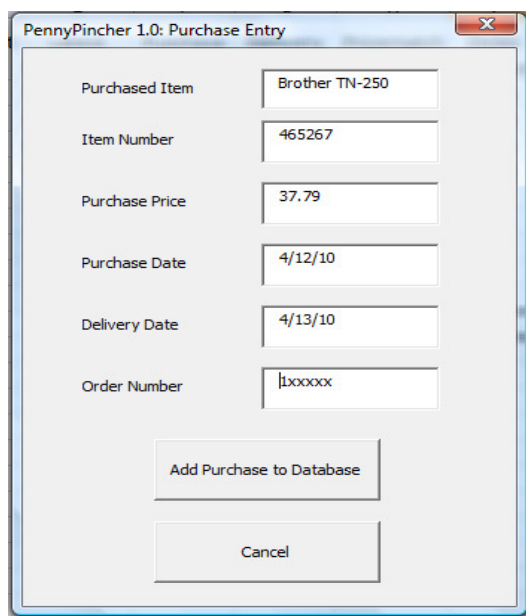
The hardest decisions for me were the ones regarding the scope of the project. When I started the project, I had intended for PennyPincher 1.0 to have many capabilities which I ultimately determined to be unrealistic. For example, I had hoped to write a program that would take invoice data from several different online retailers and check for price match opportunities amongst those retailers.

However, it quickly became clear that this would not be possible. The main reason was that each retailer's website displayed the data in a very unique way. In fact, many of the retailers I looked at displayed their prices in such a way that it was almost impossible to reliably extract the current price for use in the program.

Another factor that led to me deciding to stick to just one retailer was the horror stories I heard about trying to get online retailers to price match to a competitor. I studied several online thrift forums where I read story after story from individuals who requested, to no avail, that online retailers honor their price match policies. This made it clear that a program that attempted to automate price matching between retailers would have little success.

Ultimately, I decided to write a program that would focus solely on price matching Staples purchases to Staples prices. This was based not only what I read in the forums, but also on my personal experience. I have had several opportunities to request partial refunds from Staples as part of the firm's 14 day price guarantee, and the process has always been very easy. That was very important for a program that would make automated, e-mail requests for refunds.

Once I had determined the appropriate scope for the project, the next step was to identify the key points of data that PennyPincher 1.0 would need in order to successfully complete its task. The images below show the form that created to gather the key data points and the resulting spreadsheet.



The image shows a screenshot of a software window titled "PennyPincher 1.0: Purchase Entry". The window contains a form with several input fields and two buttons. The fields are labeled as follows:

- Purchased Item: Brother TN-250
- Item Number: 465267
- Purchase Price: 37.79
- Purchase Date: 4/12/10
- Delivery Date: 4/13/10
- Order Number: 1xxxxx

Below the input fields are two buttons: "Add Purchase to Database" and "Cancel".

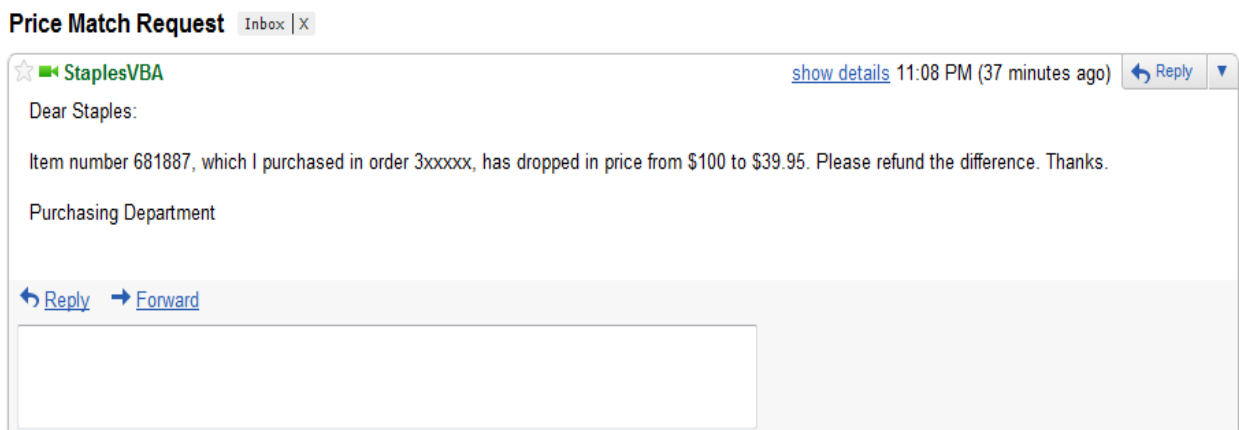
The picture above shows the key points of user input that I determined would be necessary for PennyPincher 1.0's success. In fact, the only essential points of data are the item number, the purchase price, and the order number. The other data are simply used to make the spreadsheet invoice more user friendly, as shown below.

	A	B	C	D	E	F	G	H	I
1	Item	Item Number	Purchase Price	Current Price	Latest Match Price	Purchase Date	Delivery Date	Pricematch Deadline	Order Number
2	Brother TN-250 Toner Cartridge	465267	\$37.79	\$37.79	N/A	4/12/2010	4/13/2010	4/27/2010	1xxxxx
3	Quartet® BoardGear™ Markerboard Cleaner	556134	\$7.99	\$7.99	N/A	4/5/2010	4/6/2010	4/20/2010	2xxxxx
4	Microsoft Wireless Laser Desktop 4000	681887	\$39.95	\$39.95	N/A	4/7/2010	4/8/2010	4/22/2010	3xxxxx
5	Staples® One-Touch™ 40-Sheet Adjustable 3-Hole Punch	678950	\$43.99	\$43.99	N/A	4/6/2010	4/7/2010	4/21/2010	4xxxxx
6									

You may notice a few data points in the picture above that were not listed in the user input form. These items are either generated by the program based on the other inputs, or they are downloaded from Staples.com.

As mentioned, the only points of data actually needed to accomplish the price match request are the item number, the purchase price, and the order number. The item number is essential because it is used to look up the current price information on Staples.com (more on the actual code for that process later).

The purchase price is necessary because it gives the program a starting point to which it can compare updated price information. Finally, the order number is important because PennyPincher 1.0 uses it in the price guarantee request e-mail to identify which order (and therefore which customer) requires a refund (see image of an example e-mail below).



Finally, once I had decided the scope of the project and the necessary inputs, I had to outline what steps would have to be executed in the code. After this, of course, came writing the code.

Step 2: Coding

The coding for this project took a lot of fine tuning. It was easy to take the lessons from class, such as sending e-mail with VBA, and apply them to some of the key steps in the program. However, it was very difficult getting them all to work together. Here is an over view of the basic steps executed by the program.

First, a user must input the data from Staples.com purchases. This is initiated by clicking the “Add Purchase” button seen in the image below.

	A	B	C	D	E	F	G	H	I	J	K	L
	Item	Item Number	Purchase Price	Current Price	Latest Match Price	Purchase Date	Delivery Date	Price Match Deadline	Order Number			
1												
2	Brother TN-250 Toner Cartridge	465267	\$37.79	\$37.79	N/A	4/12/2010	4/13/2010	4/27/2010	1xxxxxx			
3	Quartet® BoardGear™ Markerboard Cleaner	556134	\$7.99	\$7.99	N/A	4/5/2010	4/6/2010	4/20/2010	2xxxxxx			
4	Microsoft Wireless Laser Desktop 4000	681887	\$39.95	\$39.95	N/A	4/7/2010	4/8/2010	4/22/2010	3xxxxxx			
5	Staples® One-Touch™ 40-Sheet Adjustable 3-Hole Punch	678950	\$43.99	\$43.99	N/A	4/6/2010	4/7/2010	4/21/2010	4xxxxxx			
6												
7												
8												
9												
10												
11												
12												

Clicking this button brings up the “Purchase Entry” user form shown on page 2. The code behind this form is pretty standard. It takes the user inputs and places them into the appropriate cells in the spreadsheet.

I initially toyed with the idea of storing the info in an array within the VBA code as well, but this seemed redundant since I would still be displaying the same information for the user to see on the spreadsheet. I decided to let the spreadsheet be my array. In addition to the inputs from the user, the code inserts a “current price” amount, a “latest match price” amount, and a “price match deadline.”

The “current price” item is initially set by the program to be equal to the “purchase price.” However, every time the program runs, it downloads the current price on Staples.com and updates this cell. This is the cell that is used to determine whether or not there has been a decrease in price.

Initially, the program compares the current price to the purchase price. However, each time a price guarantee is requested, the program stores the new base price in the “latest

match price” column, which is initially defaulted to N/A (since initially no request has been made to Staples).

From the point forward, the program uses that latest price as the base to compare to the current price. This ensures that no duplicate requests are sent and that the program continues looking for new refund opportunities in case the price should drop again.

The logic used to accomplish this is displayed in the code below. The “dataRange” object is identified in a sub that runs prior to composeMail. In the case of the spreadsheet shown in previous images, it would encompass cells A2 through I5.

```
Sub composeMail()  
  
    Dim m As String  
    Dim m2 As String  
    Dim x As Long  
  
    m = "Dear Staples:" & vbNewLine & vbNewLine & "Item number <NUMBER>, which I purchased in order <ORDER>" & _  
        " has dropped in price from $<OLDPRICE> to $<NEWPRICE>." & _  
        " Please refund the difference. Thanks." & vbNewLine & vbNewLine & _  
        "Purchasing Department"  
  
    For x = 0 To row - 1  
  
        If dataRange.Cells(x + 1, 3).Value > dataRange.Cells(x + 1, 4).Value _  
            And UCase(dataRange.Cells(x + 1, 5).Value) = UCase("N/A") Then  
  
            m2 = Replace(m, "<ORDER>", dataRange.Cells(x + 1, 9).Value)  
            m2 = Replace(m2, "<OLDPRICE>", dataRange.Cells(x + 1, 3).Value)  
            m2 = Replace(m2, "<NEWPRICE>", dataRange.Cells(x + 1, 4).Value)  
            m2 = Replace(m2, "<NUMBER>", dataRange.Cells(x + 1, 2).Value)  
  
            sendMail m2, "staplesvba@gmail.com", "Price Match Request", "staplesvba", "ilovebyu"  
  
            dataRange.Cells(x + 1, 5).Value = dataRange.Cells(x + 1, 4).Value  
  
        ElseIf dataRange.Cells(x + 1, 4).Value < dataRange.Cells(x + 1, 5).Value _  
            And UCase(dataRange.Cells(x + 1, 5).Value) <> UCase("N/A") Then  
  
            m2 = Replace(m, "<ORDER>", dataRange.Cells(x + 1, 9).Value)  
            m2 = Replace(m2, "<OLDPRICE>", dataRange.Cells(x + 1, 5).Value)  
            m2 = Replace(m2, "<NEWPRICE>", dataRange.Cells(x + 1, 4).Value)  
            m2 = Replace(m2, "<NUMBER>", dataRange.Cells(x + 1, 2).Value)  
  
            sendMail m2, "staplesvba@gmail.com", "Price Match Request", "staplesvba", "ilovebyu"  
  
            dataRange.Cells(x + 1, 5).Value = dataRange.Cells(x + 1, 4).Value  
  
        End If  
    Next  
  
End Sub
```

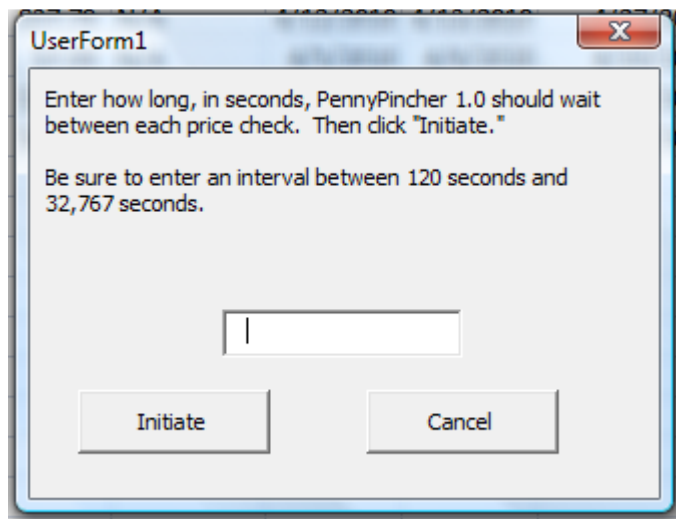
When reviewing this logic above, it may appear that one thing is missing: the 14 day deadline. My experience, and the experiences documented in the online forums I researched, shows that Staples.com often allows price matches outside the 14 day timeframe as a courtesy to their customers.

As a result, I did not write code that would determine whether or not the item in question is still within the acceptable window—after all, the window sometimes changes. Had I picked an arbitrary deadline and put it in the code, I could have caused the user to miss refunds that Staples.com would have allowed. The price match deadline mentioned on the spreadsheet was included so that the user would have a reference point in case a request was denied.

Another important point about the code above is that I used the same e-mail address to send and receive the requests. This is so that I was not bothering Staples.com with my practice requests. This part of the code would have to be updated for actual use by a business.

Also, in order to avoid giving out my personal e-mail and password, I set up a new gmail account just for this assignment.

As mentioned previously, the user can choose whether to run the program just once or on a regular basis. If the user chooses to leave the program running, he or she would simply click on the “Initiate Continual Price Check” button seen on page 4. This brings up the following user form:



The screenshot shows a dialog box titled "UserForm1" with a standard Windows-style title bar (minimize, maximize, close buttons). The main content area contains the following text:

Enter how long, in seconds, PennyPincher 1.0 should wait between each price check. Then click "Initiate."

Be sure to enter an interval between 120 seconds and 32,767 seconds.

Below the text is a single-line text input field. At the bottom of the dialog are two buttons: "Initiate" on the left and "Cancel" on the right.

As shown in the image above, the user is prompted to enter an interval of time between 120 seconds and 32,767 seconds. I chose 120 seconds based on my experience in debugging the code. Two minutes seemed to be just the right amount of time to allow for the user to step in and stop the program without crashing Excel while also running often enough to ensure that no price drops are missed. I chose 32,767 seconds as the max because it seems like more than enough time and because that corresponds for the upper limit of an integer variable. That brings me to the code used to set up this timer. This code is shown below.

```
Option Explicit
Public runTime As Double
Public runInterval As Integer
Public Const runSub = "constantCheck"

Sub initiateConstantCheck()

    frmInterval.Show

    Call constantCheck

End Sub

Sub constantCheck()

    Application.ScreenUpdating = False

    Call nameRange
    Call getPriceData
    Call composeMail
    Call runTimer

    Application.ScreenUpdating = True

End Sub

Sub runTimer()
    runTime = Now + TimeSerial(0, 0, runInterval)
    Application.OnTime EarliestTime:=runTime, Procedure:=runSub, _
        Schedule:=True
End Sub
```

While there are some steps left out of the code displayed above, it should give an idea of how the overall process works. Clicking the button on the spreadsheet calls sub `initiateConstantCheck`. This brings up `frmInterval`, which takes the user input and sets it as variable `runInterval`.

Once the code has executed the subs that populate the data, check for price changes, and send any necessary e-mails, it calls sub `runTimer`. This sets a timer which will re-run the program at the interval that was set by the user.

Difficulties

One of the biggest difficulties I encountered was working with the data from Staples.com and other sites. In fact, that was one of the factors that caused me to limit the program to just one site. Every site displayed the item prices in a different manner. In fact, even within one site, prices were displayed in different manners for different products.

This meant that I could not write one piece of code that could handle the data from several different sites. As discussed previously, I ultimately focused on Staples.com because of the reliability of their price match program, but I would have liked to include more than one site in my project.

Another difficulty I had was in setting up the project to run at regular intervals without user intervention. I couldn't find anything in the class text that assisted with this task, so I ultimately spent a lot of time researching online Excel blogs. It would have been easier, at least as far as writing the code, just to run an infinite loop.

Unfortunately, that solution would have meant that the user would have to dedicate an entire computer to this one task. Furthermore, it would have made my debugging process take a lot longer. Fortunately, I was able to find a few different solutions online. After combining pieces of each and coming up with a way to allow the user to select the interval between program runs, I had a solution that worked for me.

More importantly, I was able to learn some valuable VBA tricks that I'm sure I'll be able to use outside the classroom.