**Paper JMP-02**

# From Raw Data to Beautiful Graph Using JSL

**Michael Hecht, SAS Institute Inc., Cary, NC**

## Abstract

JSL is a powerful tool for manipulating raw data into the form needed for easy visualization in JMP. This paper presents a case study of a working script that transforms raw iOS sales data into an easy-to-interpret graph. Along the way, we learn how to summarize data, add formulas, add complex column properties, and add table scripts, all with JSL.

## Introduction

We have an iPad® app! It's called *Graph Builder*[1]*,* and it has been in the Apple App Store since March 5, 2012. Since then we have released several updates, and the app has been downloaded over 10,000 times. As they do for all iOS app developers, Apple provides us with data on app sales and downloads. Because our app is free, the sales amounts are always zero; but it is interesting to track the number of downloads. We want to take the raw download numbers that Apple provides, load them into JMP, and present them in a useful and pleasing way.

### Starting Point

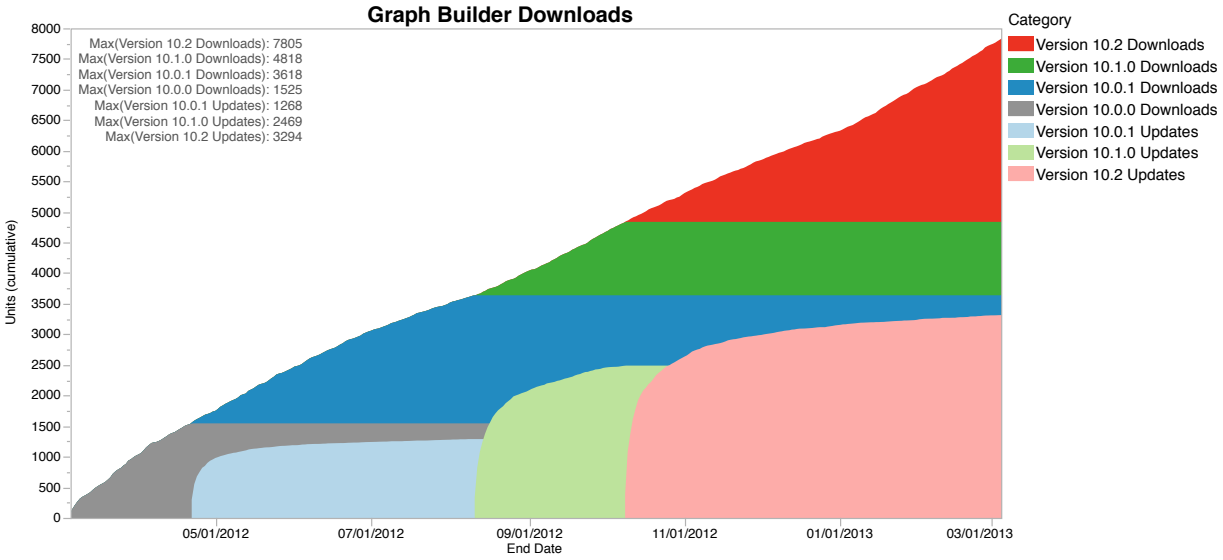
Apple's iTunes Connect website allows us to interactively download sales data for our app. They also provide a Java program called `Autoingestion`[2] which can be used to automate the data download.

Each download is a *tab-delimited text file* containing one day's worth of data. In this text data file format, each line represents a row of data; column values on the line are separated by a tab character. The first line usually contains column names. It is a common text data file format on Macintosh, and is sometimes referred to as a *tab-separated values* (or *TSV*) file.

The file names we download from Apple all have names like this: `S_D_`*`XXXXXXXX`*`_20120315.txt`. They always start with the prefix "S_", followed by "D_" for the daily data file. (There is also a weekly data file available which has a "W_" in this part of the file name.) Next comes an 8-digit number which is your Apple developer ID number (shown as eight X's in the example), followed by another underscore. Then the file names end with the date, in *YYYYMMDD* representation. The example is the name of the file with the download data for March 15, 2012.

We have a year's worth of this data: 365 text files, one for each day. We want to somehow combine them all in a way that helps us to easily visualize our download rates.

**Final Result**
Here is the final graph that our script produces.




The solid multi-color triangular area that takes up the "southeast" half of the graph shows the progression of downloads over time. The bold-colored bands represent different versions that we have released through the App Store. The lighter-colored "cliffs" in the foreground represent update downloads from users which have already downloaded a previous version. Their colors correspond to the bolder colors of first-time downloads of the same version.

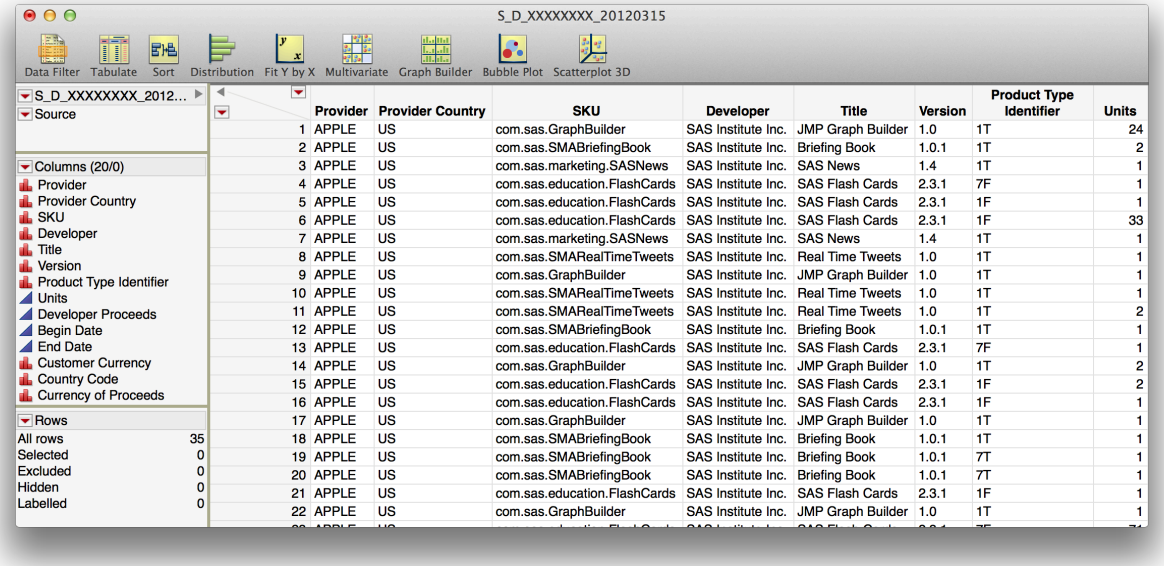
We can judge the success of a release by noticing how close, for example, the light blue cliff approaches its corresponding bold blue band. The gap between these two represent users that downloaded the first version but, for whatever reason, never upgraded it. We can easily see how this gap widens with each version.

In the top left corner, we see the actual counts for both new downloads and updates.

A JSL script automatically creates this for us each day. Let's break this script down to see how it works.

# Step 1: Importing

The first step is of course to import the raw text data files into JMP. JMP can easily import a single data file. Just open the file as *Data (Best Guess)*, and JMP does all the heavy lifting.




| | Provider | Provider Country | SKU | Developer | Title | Version | Product Type Identifier | Units |
|---|---|---|---|---|---|---|---|---|
| 1 | APPLE | US | com.sas.GraphBuilder | SAS Institute Inc. | JMP Graph Builder | 1.0 | 1T | 24 |
| 2 | APPLE | US | com.sas.SMABriefingBook | SAS Institute Inc. | Briefing Book | 1.0.1 | 1T | 2 |
| 3 | APPLE | US | com.sas.marketing.SASNews | SAS Institute Inc. | SAS News | 1.4 | 1T | 1 |
| 4 | APPLE | US | com.sas.education.FlashCards | SAS Institute Inc. | SAS Flash Cards | 2.3.1 | 7F | 1 |
| 5 | APPLE | US | com.sas.education.FlashCards | SAS Institute Inc. | SAS Flash Cards | 2.3.1 | 1F | 1 |
| 6 | APPLE | US | com.sas.education.FlashCards | SAS Institute Inc. | SAS Flash Cards | 2.3.1 | 1F | 33 |
| 7 | APPLE | US | com.sas.marketing.SASNews | SAS Institute Inc. | SAS News | 1.4 | 1T | 1 |
| 8 | APPLE | US | com.sas.SMARealTimeTweets | SAS Institute Inc. | Real Time Tweets | 1.0 | 1T | 1 |
| 9 | APPLE | US | com.sas.GraphBuilder | SAS Institute Inc. | JMP Graph Builder | 1.0 | 1T | 1 |
| 10 | APPLE | US | com.sas.SMARealTimeTweets | SAS Institute Inc. | Real Time Tweets | 1.0 | 1T | 1 |
| 11 | APPLE | US | com.sas.SMARealTimeTweets | SAS Institute Inc. | Real Time Tweets | 1.0 | 1T | 2 |
| 12 | APPLE | US | com.sas.SMABriefingBook | SAS Institute Inc. | Briefing Book | 1.0.1 | 1T | 1 |
| 13 | APPLE | US | com.sas.education.FlashCards | SAS Institute Inc. | SAS Flash Cards | 2.3.1 | 7F | 1 |
| 14 | APPLE | US | com.sas.GraphBuilder | SAS Institute Inc. | JMP Graph Builder | 1.0 | 1T | 2 |
| 15 | APPLE | US | com.sas.education.FlashCards | SAS Institute Inc. | SAS Flash Cards | 2.3.1 | 1F | 2 |
| 16 | APPLE | US | com.sas.education.FlashCards | SAS Institute Inc. | SAS Flash Cards | 2.3.1 | 1F | 1 |
| 17 | APPLE | US | com.sas.GraphBuilder | SAS Institute Inc. | JMP Graph Builder | 1.0 | 1T | 1 |
| 18 | APPLE | US | com.sas.SMABriefingBook | SAS Institute Inc. | Briefing Book | 1.0.1 | 1T | 1 |
| 19 | APPLE | US | com.sas.SMABriefingBook | SAS Institute Inc. | Briefing Book | 1.0.1 | 7T | 1 |
| 20 | APPLE | US | com.sas.SMABriefingBook | SAS Institute Inc. | Briefing Book | 1.0.1 | 7T | 1 |
| 21 | APPLE | US | com.sas.education.FlashCards | SAS Institute Inc. | SAS Flash Cards | 2.3.1 | 1F | 1 |
| 22 | APPLE | US | com.sas.GraphBuilder | SAS Institute Inc. | JMP Graph Builder | 1.0 | 1T | 1 |

JMP already understands the tab-delimited text file format, and it automatically names the columns for us appropriately. Notice that the data file contains download information for all of SAS Institute's iOS apps — not just for Graph Builder, which we're interested in. Also notice the **Product Type Identifier** column. This coded column distinguishes between new downloads and update downloads from a previous version.

JMP also adds a **Source** table property. This is a JSL description of the operation that produced this data table. It's very handy for us because we can use it as a starting point for our script.

```
Open(
	"$DESKTOP/iOS Sales/Data/S_D_XXXXXXXX_20120315.txt",
	columns(
		Column( "Provider", Character, Nominal ),
		Column( "Provider Country", Character, Nominal ),
		Column( "SKU", Character, Nominal ),
		Column( "Developer", Character, Nominal ),
		Column( "Title", Character, Nominal ),
		Column( "Version", Character, Nominal ),
		Column( "Product Type Identifier", Character, Nominal ),
		Column( "Units", Numeric, Continuous, Format( "Best", 10 ) ),
...
	),
	Import Settings(
		End Of Line( CRLF, CR, LF ),
		End Of Field( Tab, Comma, CSV( 1 ) ),
		Strip Quotes( 1 ),
		Use Apostrophe as Quotation Mark( 0 ),
		Scan Whole File( 1 ),
		Treat empty columns as numeric( 0 ),
		CompressNumericColumns( 0 ),
		CompressCharacterColumns( 0 ),
		CompressAllowListCheck( 0 ),
...
	)
)
```

When run, we get the exact same result as when we performed the operation interactively. Actually, this script is a bit verbose. I've already trimmed out parts of it, but we can reduce it down to just this, and let JMP use its own defaults for the rest.

```
// Import one file
Open(
	"Data/S_D_XXXXXXXX_20120315.txt",
	Columns(
		Column( "Provider", Character, Nominal ),
		Column( "Provider Country", Character, Nominal ),
		Column( "SKU", Character, Nominal ),
		Column( "Developer", Character, Nominal ),
		Column( "Title", Character, Nominal ),
		Column( "Version", Character, Nominal ),
	),
	Import Settings( End Of Field( Tab ))
);
```

There are a few things to note here. First, we save this script in the same directory as our "Data" directory. That allows us to refer to our data files by a relative path. Second, we provide explicit descriptions for the first six columns, because in some input files, JMP has trouble determining on its own that they should be character columns.

So far, our script works fine for a single data file, but we want to import *all* the data files. So we need to put a loop around our script. But how will we loop through all the files in our "Data" directory? Simple: we use JSL's **Files In Directory()** function!

```
// Import all files
files = Files In Directory( "Data" );
For( i = 1, i <= N Items( files ), i++,
	f = files[ i ];

	// Skip if not a data file
	If( !Ends With( f, ".txt" ), Continue());
	If( !Starts With( f, "S_D_" ), Continue());

	Write( Eval Insert( "\!N^i^: Importing ^f^" ));
	Open( Eval Insert( "Data/^f^" ),
		...
	);
);
```

On the first line, **Files In Directory()** returns a list of all the file names within the "Data" directory. We can easily iterate through this list with a **For()** loop on line 2.

Inside the **For()**, we want to skip past files that may be in that directory, but aren't data files that we want to import. The first **If()** looks for files whose names don't end with ".txt". When a file name like this is encountered, we use the **Continue()** function, which starts us back at the top of the **For()**, for the next value of *i*. The second **If()** filters out files whose names don't start with "S_D_", in the same fashion.

Next, we write a message to the log, just to help us see what's going on; then we import the $i^{th}$ file, as before. Both of these operations use the handy **Eval Insert()** function. This function returns its input string, after it evaluates each expression delimited by the caret symbols (^) and replaces that text with the result. So ^*i*^ becomes the value of *i* for each loop iteration, and ^*f*^ becomes the file name. We follow the JMP convention of writing each line to the log starting with a newline sequence ("\!N").

This does the job — sort of. When we run it, we get 365 data table windows, one for each data file in the year's worth of data we're importing. We really need to concatenate all these data files into a single data table; and we don't need to see all the intermediate data tables in the process!

The first data file we import should become the beginning of our final data table. All others imported after that should be concatenated to it. So we need a way to distinguish the first table from the rest. We could just use **If(** *i* == 1, ... **)**, but that won't work if the first file in the directory is one we want to skip over. Here's a better solution.

```
// Import all files and concatenate them together
dt = Empty();

files = Files In Directory( "Data" );
For( i = 1, i <= N Items( files ), i++,
	f = files[ i ];
	If( !Ends With( f, ".txt" ), Continue());
	If( !Starts With( f, "S_D_" ), Continue());

	Write( Eval Insert( "\!N^i^: Importing ^f^" ));
	adt = Open( Eval Insert( "Data/^f^" ),
		Columns(
			...
		),
		Import Settings( End Of Field( Tab )),
		Invisible
	);

	If( Is Empty( dt ),
		// First import
		dt = adt;
	,
		// All subsequent imports
		dt << Concatenate( adt, Append to first table );
		Close( adt, No Save );
	);
);

// Save the combined data file
dt << Delete Table Property( "Source" );
dt << Set Name( "iOS Sales" );
dt << Save( "iOS Sales.jmp" );

dt << New Data View;	// For debugging only
```

How does this work? We start by creating a variable *dt*, which will hold the reference to our final data table. We need to initialize it to a value that lets us distinguish the first import from the ones to follow; so we use the **Empty()** function.

Each import places the reference to the imported data table in the variable *adt*. Note the addition of the *Invisible* option on the **Open()**. That option tells JMP to not go to the effort to create a window for this data table. It imports the data and creates all the usual data table structures in memory, it just doesn't actually show us the table in a window. This actually speeds up the script's execution quite a bit!

The next **If()** determines whether this is the first import or a subsequent import. It doesn't work to use **If(** *dt* == **Empty()**, ... **)**. Instead, we must use the **Is Empty()** function to test for that. If true, this is the first import so we just move *adt* to *dt*.

The second half of the **If()** is used for all imports after the first one, because now *dt* is no longer empty. For these imports, we use the **<< Concatenate** message to append a copy of the data rows to the first import. After that, we don't need the data table we just imported, so we close it.

Outside the loop, *dt* holds a reference to the data table that has all of our combined data in it. We do some final fix-up on the table: we delete its **Source** property, which will be the **Open()** command that imported the initial data file. We also set the data table's name; we don't want it to have some crazy name like "S_D_*blah_blah_blah*"! Then we save the table to disk for future reference.

But it's still invisible. So we send it the **<< New Data View** message to tell JMP that it's time to make a window. This is just a temporary step for debugging, so we can see our progress along the way.

## Step 2: Making a Subset

We now have all of our raw data stored in one combined JMP data table. But this data table contains download information for all SAS Institute apps. We would like to restrict it to just the data that pertains to our Graph Builder app. Interactively, we could do this with the Distribution platform; but from JSL, our best choice is to use the **<< Subset** message.

```
// Make a subset of just the Graph Builder sales
dt << Select Where( :SKU == "com.sas.GraphBuilder" );
dt subset = dt << Subset( Selected Rows( 1 ), Invisible );
dt << Clear Row States;

// Done with combined data table
Close( dt, No Save );

dt subset << Delete Table Property( "Source" );
dt subset << Set Name( "iOS Sales (Graph Builder)" );
dt subset << Save( "iOS Sales (Graph Builder).jmp" );

dt subset << New Data View;      // For debugging only
```

In JSL, making a subset is a two-step process. First, we select all the rows that we want to include in the subset; then, we construct a subset of the selected rows. The **<< Select Where**

message accomplishes the first part. We send it to our combined data table, and it selects only the rows where the *SKU* column has the value “com.sas.GraphBuilder”, which is the unique identifier for our application on the App Store. We use the SKU instead of the app name, because our app’s SKU will stay the same even if we decide to rename our app.

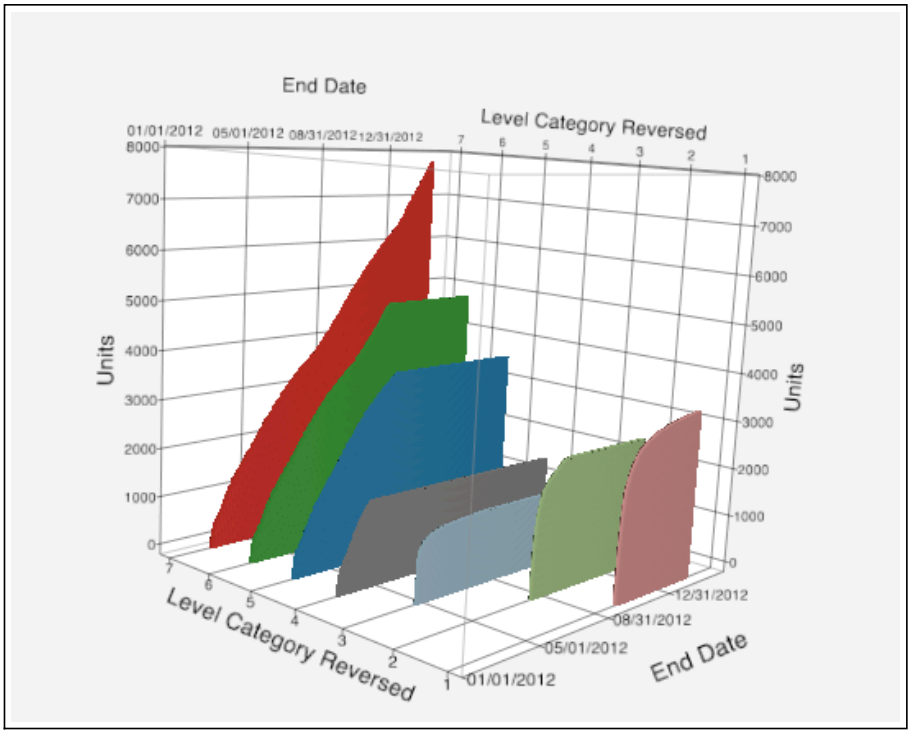
Next, we send our combined data table the **<< Subset** message. This message returns a reference to a new data table containing the subset. The *Selected Rows(* 1 *)* argument does not mean to only use one selected row. The 1 means “yes, use selected rows”. Note the continued use of the *Invisible* option, to prevent JMP from making windows for these data tables before we’re ready for them.

Finally, we send our combined data table the **<< Clear Row States** message. This deselects the rows we just selected. It’s not strictly necessary in this case, but if you need to do it, this is how it’s done. At this point, we’re finished with the combined data table so we close it.

We can now do a little fix-up on the subset data table that we still hold a reference to. As before, we delete its **Source** property (this one will be the **<< Subset** message), name it, and save it to disk.

## Step 3: Summarizing

We have now extracted the portion of the data we want to graph. But we need to reshape it into a format that allows us to create the graph we want. We want a Graph Builder area graph with the “new download” categories stacked. But at the same time, we want the “update download” categories overlaid. If we could look at it in 3D, we can see how we trick Graph Builder into doing this.

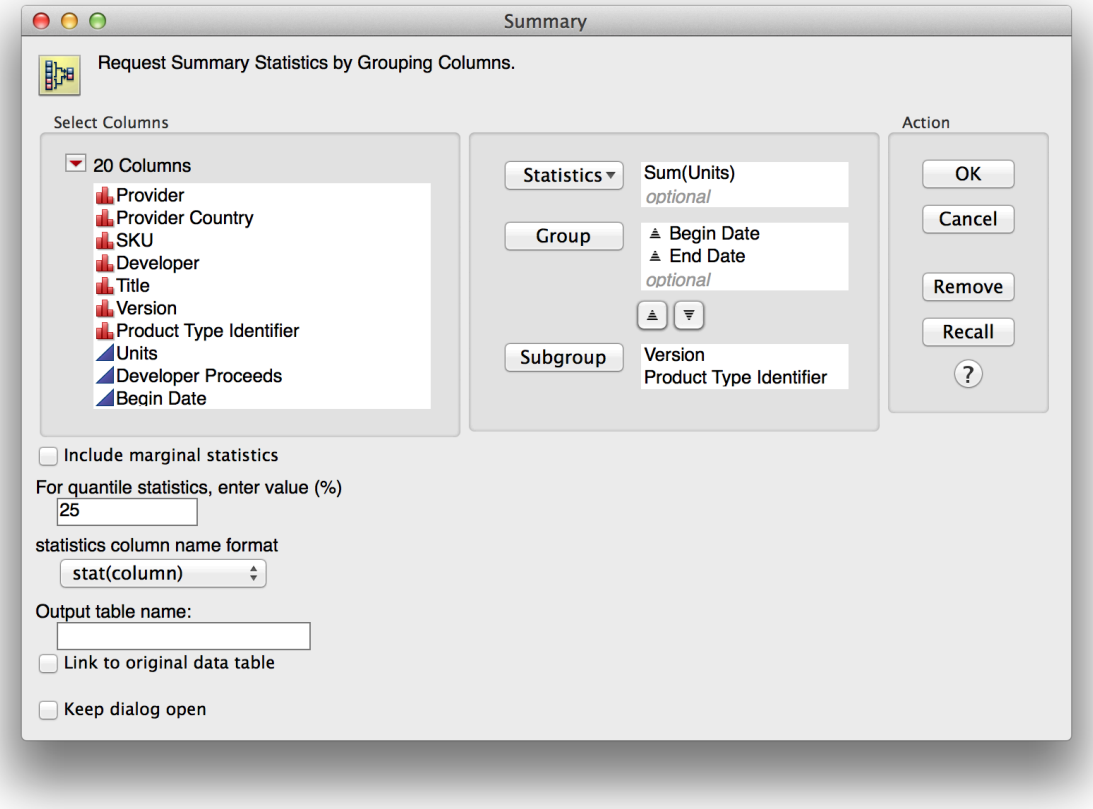


This view informs us how to reshape the data:

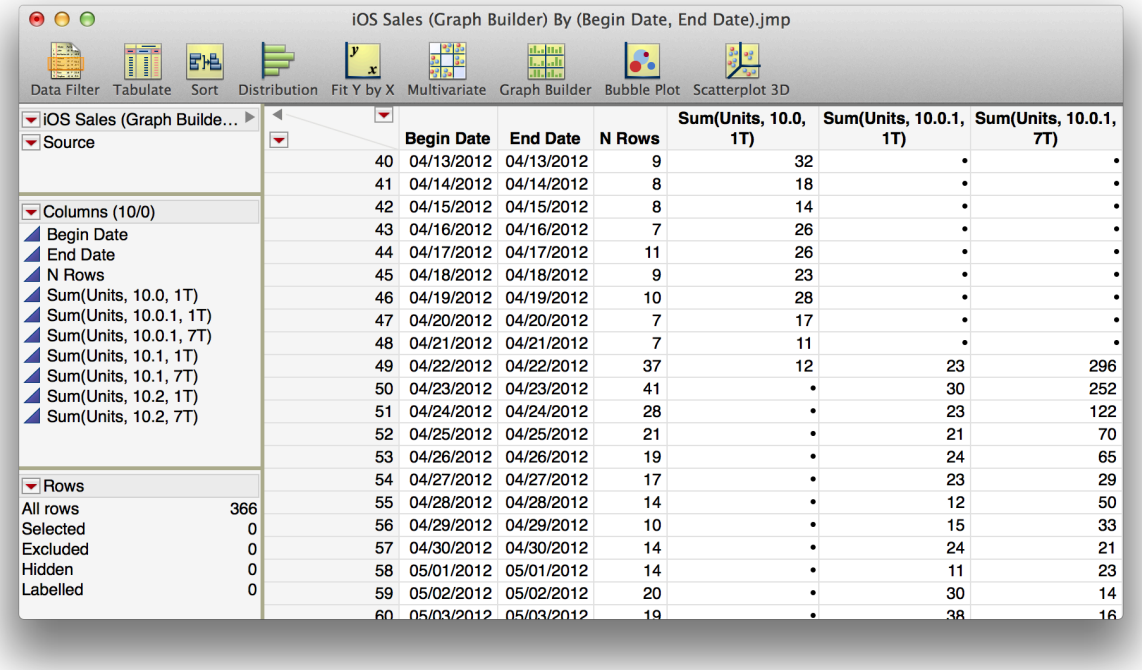
- We need an overlay variable with a separate value for each overlaid version and download method.
- The Y variable is the cumulative units.
- When a new version is introduced, the units for the current version stay constant at their maximum value.
- Unit values before a version is introduced just have the same value as their prior version. But you can't see them because the prior version is overlaid precisely on top. So they appear to be stacked.

Our overlay categories need to be a combination of the *Version* column and the *Product Identifier Type* column. Apple's documentation for their sales data format describes all the values that *Product Identifier Type* can have. But for our purposes, we need only be concerned with "1T", which means "new download" and "7F", which means "update download". For example, Version 10.0.1 has both new and update downloads, so we will need categories for "10.0.1, 1T" and "10.0.1, 7F".

Our subset data also has multiple rows for each day, corresponding to the sales in separate countries. We want to combine these rows together; so we have a single row for each day, with separate values for each *Version/Product Identifier Type* combination. We can use JMP's **Summary** command to do all of this. Here's how we set up the command interactively.



We will compute the sum of all the *Units* downloaded on a given day. So we must group by *Begin/End Date*. (They are always the same date.) Using the Subgroup feature, we can specify that separate Unit counts should be computed for each unique combination of *Version* and *Product Type Identifier*. The resulting table looks like this.




| | Begin Date | End Date | N Rows | Sum(Units, 10.0, 1T) | Sum(Units, 10.0.1, 1T) | Sum(Units, 10.0.1, 7T) |
|---|---|---|---|---|---|---|
| 40 | 04/13/2012 | 04/13/2012 | 9 | 32 | • | • |
| 41 | 04/14/2012 | 04/14/2012 | 8 | 18 | • | • |
| 42 | 04/15/2012 | 04/15/2012 | 8 | 14 | • | • |
| 43 | 04/16/2012 | 04/16/2012 | 7 | 26 | • | • |
| 44 | 04/17/2012 | 04/17/2012 | 11 | 26 | • | • |
| 45 | 04/18/2012 | 04/18/2012 | 9 | 23 | • | • |
| 46 | 04/19/2012 | 04/19/2012 | 10 | 28 | • | • |
| 47 | 04/20/2012 | 04/20/2012 | 7 | 17 | • | • |
| 48 | 04/21/2012 | 04/21/2012 | 7 | 11 | • | • |
| 49 | 04/22/2012 | 04/22/2012 | 37 | 12 | 23 | 296 |
| 50 | 04/23/2012 | 04/23/2012 | 41 | • | 30 | 252 |
| 51 | 04/24/2012 | 04/24/2012 | 28 | • | 23 | 122 |
| 52 | 04/25/2012 | 04/25/2012 | 21 | • | 21 | 70 |
| 53 | 04/26/2012 | 04/26/2012 | 19 | • | 24 | 65 |
| 54 | 04/27/2012 | 04/27/2012 | 17 | • | 23 | 29 |
| 55 | 04/28/2012 | 04/28/2012 | 14 | • | 12 | 50 |
| 56 | 04/29/2012 | 04/29/2012 | 10 | • | 15 | 33 |
| 57 | 04/30/2012 | 04/30/2012 | 14 | • | 24 | 21 |
| 58 | 05/01/2012 | 05/01/2012 | 14 | • | 11 | 23 |
| 59 | 05/02/2012 | 05/02/2012 | 20 | • | 30 | 14 |
| 60 | 05/03/2012 | 05/03/2012 | 19 | • | 38 | 16 |

We now have a single row for each date. We also have a separate column for each *Version/ Product Type Identifier* combination. The data values in those columns are the number of downloads of that category. For example, we can see that on April 22, Version 10.0.1 was released. On that day, in the hours before its release, there were 12 new downloads of Version 10.0. After Version 10.0.1 became available, there were 23 downloads of the new version and 296 updates from users who already had Version 10.0.

As before, the **Source** table property describes what we just did in script form.

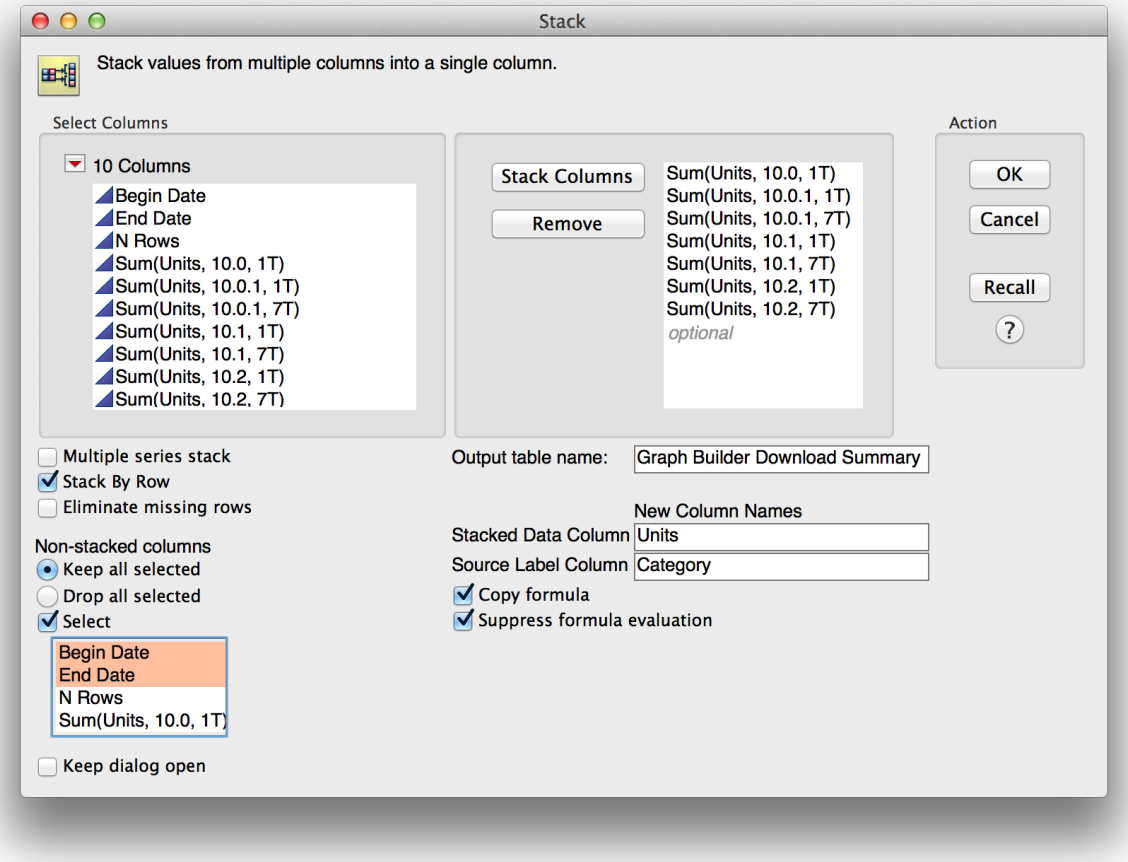
```
Data Table( "iOS Sales (Graph Builder)" ) << Summary(
      Group( :Begin Date, :End Date ),
      Sum( :Units ),
      Subgroup( :Version, :Product Type Identifier ),
      Link to original data table( 0 )
)
```

We can add this to the script we are constructing, changing the source table to the reference we already have, and adding the usual *Invisible* option. The **<< Summary** message returns a reference to the summary data table it creates, so we want to capture that.

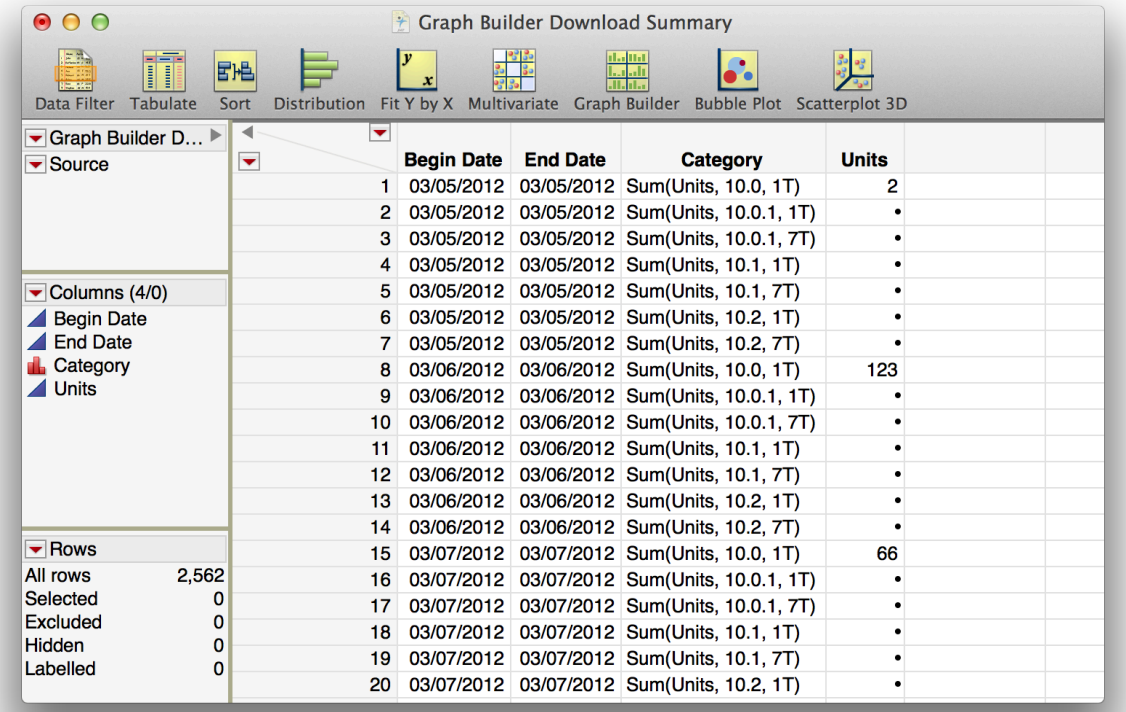
```
// Summarize the Graph Builder sales data
dt temp = dt subset << Summary(
       Group( :Begin Date, :End Date ),
       Sum( :Units ),
       Subgroup( :Version, :Product Type Identifier ),
       Link to original data table( 0 ),
       Invisible
);

// Done with Graph Builder subset
Close( dt subset, No Save );

dt temp << New Data View;  // For debugging only
```

## Step 4: Stacking

So far, we've managed to split up our data into the categories we need. But to produce the graph we want, we really need all of the units to be in a single column. And we need another column of just the categories. So we need to **Stack** all our Sum(*Units, ...* columns together, like so.



This gives us a new data table with our data reshaped.




| | Begin Date | End Date | Category | Units |
|---|---|---|---|---|
| 1 | 03/05/2012 | 03/05/2012 | Sum(Units, 10.0, 1T) | 2 |
| 2 | 03/05/2012 | 03/05/2012 | Sum(Units, 10.0.1, 1T) | • |
| 3 | 03/05/2012 | 03/05/2012 | Sum(Units, 10.0.1, 7T) | • |
| 4 | 03/05/2012 | 03/05/2012 | Sum(Units, 10.1, 1T) | • |
| 5 | 03/05/2012 | 03/05/2012 | Sum(Units, 10.1, 7T) | • |
| 6 | 03/05/2012 | 03/05/2012 | Sum(Units, 10.2, 1T) | • |
| 7 | 03/05/2012 | 03/05/2012 | Sum(Units, 10.2, 7T) | • |
| 8 | 03/06/2012 | 03/06/2012 | Sum(Units, 10.0, 1T) | 123 |
| 9 | 03/06/2012 | 03/06/2012 | Sum(Units, 10.0.1, 1T) | • |
| 10 | 03/06/2012 | 03/06/2012 | Sum(Units, 10.0.1, 7T) | • |
| 11 | 03/06/2012 | 03/06/2012 | Sum(Units, 10.1, 1T) | • |
| 12 | 03/06/2012 | 03/06/2012 | Sum(Units, 10.1, 7T) | • |
| 13 | 03/06/2012 | 03/06/2012 | Sum(Units, 10.2, 1T) | • |
| 14 | 03/06/2012 | 03/06/2012 | Sum(Units, 10.2, 7T) | • |
| 15 | 03/07/2012 | 03/07/2012 | Sum(Units, 10.0, 1T) | 66 |
| 16 | 03/07/2012 | 03/07/2012 | Sum(Units, 10.0.1, 1T) | • |
| 17 | 03/07/2012 | 03/07/2012 | Sum(Units, 10.0.1, 7T) | • |
| 18 | 03/07/2012 | 03/07/2012 | Sum(Units, 10.1, 1T) | • |
| 19 | 03/07/2012 | 03/07/2012 | Sum(Units, 10.1, 7T) | • |
| 20 | 03/07/2012 | 03/07/2012 | Sum(Units, 10.2, 1T) | • |

And we can once again use the handy **Source** property to see how our interactive work is expressed in JSL.

```
Data Table( "iOS Sales (Graph Builder) By (Begin Date, End Date).jmp" ) <<
Stack(
	columns(
		:Name( "Sum(Units, 10.0, 1T)" ),
		:Name( "Sum(Units, 10.0.1, 1T)" ),
		:Name( "Sum(Units, 10.0.1, 7T)" ),
		:Name( "Sum(Units, 10.1, 1T)" ),
		:Name( "Sum(Units, 10.1, 7T)" ),
		:Name( "Sum(Units, 10.2, 1T)" ),
		:Name( "Sum(Units, 10.2, 7T)" )
	),
	Source Label Column( "Category" ),
	Stacked Data Column( "Units" ),
	Name( "Non-stacked columns" )(Keep( :Begin Date, :End Date )),
	Output Table( "Graph Builder Download Summary" )
)
```

But now we have a problem! We don't know *a priori* how many *Version/Product Type Identifier* categories to expect, so how are we supposed to we build the *Columns()* argument? And when we release the next version of our app, we would like our graph to simply adjust without us having to edit the script. Can we code our script in such a way that it automatically determines how many columns to stack?

Yes we can! Here is a JSL snippet that loops through all the columns in *dt temp*, and determines which ones are *Units* columns.

```
// Locate all Unit columns
For( c = 1, c <= N Cols( dt temp ), c++,
	col = Column( dt temp, c );
	col name = col << Get Name;
	col words = Words( col name, "," );
	If( col words[ 1 ] != "Sum(Units", Continue());

	Write( Eval Insert( "\!N^c^: ^col words^" ));
);
```

For each column, this script gets its name, then uses the **Words()** function to break up the column name into "words" separated by commas. The rest of the loop only considers columns whose name starts with the "word" *"Sum(Units"*. This gives us just the columns we want, as we can see on the JMP Log.

```
4: {"Sum(Units", " 10.0", " 1T)"}
5: {"Sum(Units", " 10.0.1", " 1T)"}
6: {"Sum(Units", " 10.0.1", " 7T)"}
7: {"Sum(Units", " 10.1", " 1T)"}
8: {"Sum(Units", " 10.1", " 7T)"}
9: {"Sum(Units", " 10.2", " 1T)"}
10: {"Sum(Units", " 10.2", " 7T)"}
```

So instead of writing them to the log, let's collect their column references into a list. We can then pass that list to the **<< Stack** message, and it will automatically stack all the unit columns for us.

```
stack cols = {};

// Collect all Unit columns
For( c = 1, c <= N Cols( dt temp ), c++,
	col = Column( dt temp, c );
	col name = col << Get Name;
	col words = Words( col name, "," );
	If( col words[ 1 ] != "Sum(Units", Continue());

	Insert Into( stack cols, Column( dt temp, col name ));
);

// Stack the Units; create the categories
dt summary = dt temp << Stack(
	Columns( stack cols ),
	Output Table Name( "Graph Builder Download Summary" ),
	Source Label Column( "Category" ),
	Stacked Data Column( "Units" ),
	Name( "Non-stacked columns" )( Keep( :Begin Date, :End Date ))
);
```

Running this script gives us the same stacked data table we were able to create interactively. But we didn't need to explicitly name all the *Units* columns — the script found them for us automatically!

## Step 5: Computing Cumulative Units

So far, we've managed to import our data, combine it together, subset only the part we want, and reshape it into a form ready for graphing. Unfortunately, it's not really the data that we want to graph. We have the day-to-day download numbers, but we want to see cumulative download numbers.

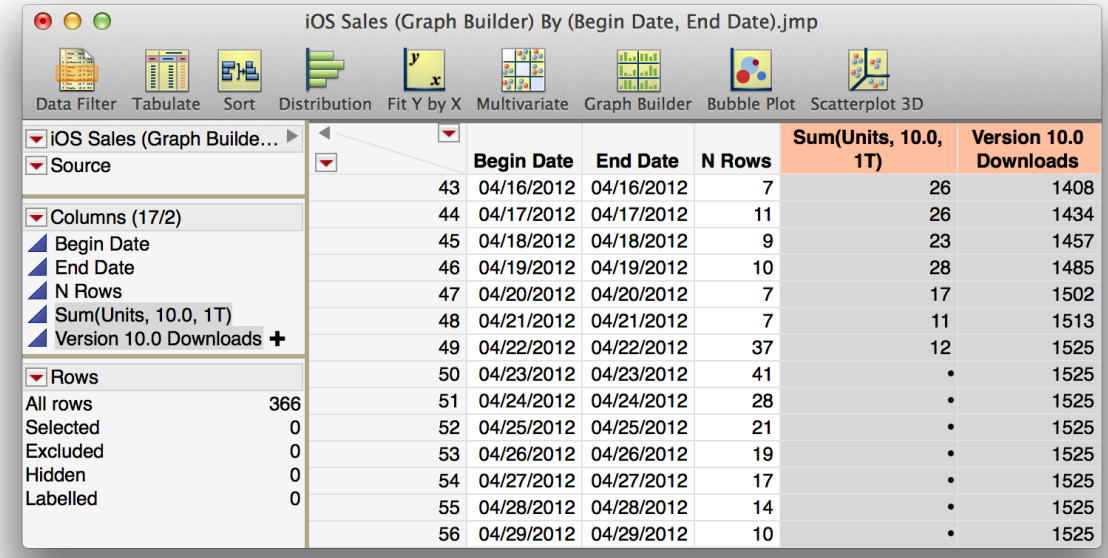
We can compute a cumulative unit number with a column formula. This formula will be much easier to build on the unstacked data, so let's back up a step to before our data was stacked. We can add formula columns to compute cumulative units to the unstacked data table, then stack the formula columns instead of the day-to-day columns.

**Preparation**

What these formulas look like? We have a column for *Sum(Units, 10.0, 1T)* with the day-to-day numbers for new downloads of Version 10.0. We will create a new column named *Version 10.0 Downloads* with this column formula.

$$\text{Sum}(\ \textit{Sum(Units, 10.0, 1T)}, \text{Lag}(\ \textit{Version 10.0 Downloads}, 1\ ))$$

This is your basic cumulative column formula. It sums together the current row's day-to-day download number with the previous row's cumulative download number. The result looks like this.




| | Begin Date | End Date | N Rows | Sum(Units, 10.0, 1T) | Version 10.0 Downloads |
|---|---|---|---|---|---|
| 43 | 04/16/2012 | 04/16/2012 | 7 | 26 | 1408 |
| 44 | 04/17/2012 | 04/17/2012 | 11 | 26 | 1434 |
| 45 | 04/18/2012 | 04/18/2012 | 9 | 23 | 1457 |
| 46 | 04/19/2012 | 04/19/2012 | 10 | 28 | 1485 |
| 47 | 04/20/2012 | 04/20/2012 | 7 | 17 | 1502 |
| 48 | 04/21/2012 | 04/21/2012 | 7 | 11 | 1513 |
| 49 | 04/22/2012 | 04/22/2012 | 37 | 12 | 1525 |
| 50 | 04/23/2012 | 04/23/2012 | 41 | • | 1525 |
| 51 | 04/24/2012 | 04/24/2012 | 28 | • | 1525 |
| 52 | 04/25/2012 | 04/25/2012 | 21 | • | 1525 |
| 53 | 04/26/2012 | 04/26/2012 | 19 | • | 1525 |
| 54 | 04/27/2012 | 04/27/2012 | 17 | • | 1525 |
| 55 | 04/28/2012 | 04/28/2012 | 14 | • | 1525 |
| 56 | 04/29/2012 | 04/29/2012 | 10 | • | 1525 |

Because we use the **Sum()** function, the cumulative download number stays at its maximum value after we run out of day-to-day numbers (i.e. when a new version is released).

What about Version 10.0.1 — what should its column formula look like? It's similar to the one we just created, but it should include the downloads from the prior version as well.

```
Sum(
        Sum(Units, 10.0, 1T),
        Sum(Units, 10.0.1, 1T),
        Lag( Version 10.0.1 Downloads, 1 )
)
```

We do this so that in our graph, these categories will appear to be stacked on top of each other, even though we are using an overlay. In general, each cumulative formula for downloads follows this pattern:

```
Sum(
        <day-to-day downloads for all versions, up to and including this one>,
        Lag( this column, 1 )
)
```

We need to construct cumulative columns for the update units as well. But since they won't appear stacked in our graph, they can use the basic formula.

```
Sum(
        <day-to-day update downloads for this version only>,
        Lag( this column, 1 )
)
```

Once we've created all these new cumulative columns, we can stack them instead of the day-to-day columns. Their column names will become our categories.

**Coding**

So how do we construct these new columns and their formulas in JSL?

We can start by creating names for them. Let's go back to our column collection loop and add some code to construct a column name for each formula column. We'll call it *form col name.*

```
// Add cumulative columns
For( c = 1, c <= N Cols( dt temp ), c++,
	col = Column( dt temp, c );
	col name = col << Get Name;
	col words = Words( col name, "," );
	If( col words[ 1 ] != "Sum(Units", Continue());

	ver = Trim( col words[ 2 ]);
	act code = col words[ 3 ];
	Match( act code,
		" 1T)", action = "Downloads",
		" 7T)", action = "Updates",
		Throw()
	);

	form col name = Eval Insert( "Version ^ver^ ^action^" );

	Write( Eval Insert( "\!N^c^: ^form col name^" ));
);
```

Note the use of **Throw()**, in case Apple gives us a *Product Type Identifier* we don't expect. Running this script puts the following on the JMP Log. So far, so good.

```
4: Version 10.0 Downloads
5: Version 10.0.1 Downloads
6: Version 10.0.1 Updates
7: Version 10.1 Downloads
8: Version 10.1 Updates
9: Version 10.2 Downloads
10: Version 10.2 Updates
```

Now we need to make a new column each time through the loop, and give it a formula. We'd like to use the same technique we learned to make the list of stacking columns here. But because of the complexity of building and adding formulas, we need a different method. We will do this by building a string that has the JSL to add the column. Then we can parse and evaluate the string to make it actually happen.

Each formula is just a **Sum()** function, but they have different arguments. If we can build the arguments in a string, then we can add the column and its formula with this code. Note that in our script, the JSL string is all on a single line.

```
// Add cumulative columns
For( c = 1, c <= N Cols( dt temp ), c++,
	...

	set formula str = Eval Insert( "dt temp << New Column( \!"^form col name^\!",
Numeric, Continuous, Formula( Sum( ^sum arg str^ )))" );
	Eval( Parse( set formula str ));
);
```

There are a few things to be aware of here: first, our string of JSL must necessarily have other strings inside it. To make this work, we need to write the embedded string's quotes in escaped form: \!". Second, once we have built *set formula str* — our string containing the JSL we want to

perform — we use **Eval( Parse())** to make it happen. **Parse()** converts our string into a JSL expression. **Eval()** evaluates that expression, performing the action it describes.

Now we just need to build *sum arg str*. Let's start by making a list of the columns we want to sum. For downloads, that will be a list of all the columns we've processed so far. For updates, it's just the one column we're currently processing. We can express that in JSL this way:

```
download cols = {};

// Add cumulative columns
For( c = 1, c <= N Cols( dt temp ), c++,
	...

	sum cols = {};
	Match( action,
	"Downloads",
		Insert Into( download cols, col );
		sum cols = download cols;

	,
	"Updates",
		Insert Into( sum cols, col );
	);

	Write( Eval Insert( "\!N^c^: ^form col name^: ^sum cols^" ));
);
```

This logic is a bit complicated to follow. But basically, we are creating a list of all the download columns, named *download cols*. Each column we process with the "Downloads" action, we insert into this list. That's the list we will pass to **Sum()**, so we move it to *sum cols*. For update columns, we simply sum the column itself, so there we set *sum cols* to a list of only one item.

Here's the JMP Log output from the above script.

```
4: Version 10.0 Downloads: {Column("Sum(Units, 10.0, 1T)")}
5: Version 10.0.1 Downloads: {Column("Sum(Units, 10.0, 1T)"), Column("Sum(Units,
10.0.1, 1T)")}
6: Version 10.0.1 Updates: {Column("Sum(Units, 10.0.1, 7T)")}
7: Version 10.1 Downloads: {Column("Sum(Units, 10.0, 1T)"), Column("Sum(Units,
10.0.1, 1T)"), Column("Sum(Units, 10.1, 1T)")}
8: Version 10.1 Updates: {Column("Sum(Units, 10.1, 7T)")}
9: Version 10.2 Downloads: {Column("Sum(Units, 10.0, 1T)"), Column("Sum(Units,
10.0.1, 1T)"), Column("Sum(Units, 10.1, 1T)"), Column("Sum(Units, 10.2, 1T)")}
10: Version 10.2 Updates: {Column("Sum(Units, 10.2, 7T)")}
```

Notice that each "Downloads" column gets a progressively longer list, but the "Updates" columns have lists with only one item.

We now have the list of column references we want to sum in the variable *sum cols*. (We also want to sum the **Lag(** *current column* **)**, but we'll add that in a minute.) We just need to get this list into string form so we can insert it into our formula. We'll do that by making an intermediate list that has each column's name as a string. That is, we want to convert this:

```
{ Column( a ), Column( b ), Column( c )}
```

to this:

```
{ ":a", ":b", ":c" }
```

Here is some code to do it.

```
sum col strs = {};
For( i = 1, i <= N Items( sum cols ), i++,
	Insert Into( sum col strs, Eval Insert( ":Name( \!"^sum cols[ i ] << Get
Name^\!" )" ));
);
```

This code loops through all the items of *sum cols*, and inserts a corresponding string into *sum col strs*. The string is built using our old friend **Eval Insert()**. It has this form: ":Name( "^*name of this col*^" )".

Again we use of the special escape sequence \!" to embed double quote characters in our string. Second, we must use the **Name()** operator to build the column name. That's because our column names have parentheses and commas, which would otherwise confuse JMP when we ask it to parse the string we're building. Finally, we're using a more complex expression within the carets of our **Eval Insert()** this time. We actually send the **<< Get Name** message to the column reference within the *sum cols* list directly. Pretty powerful stuff!

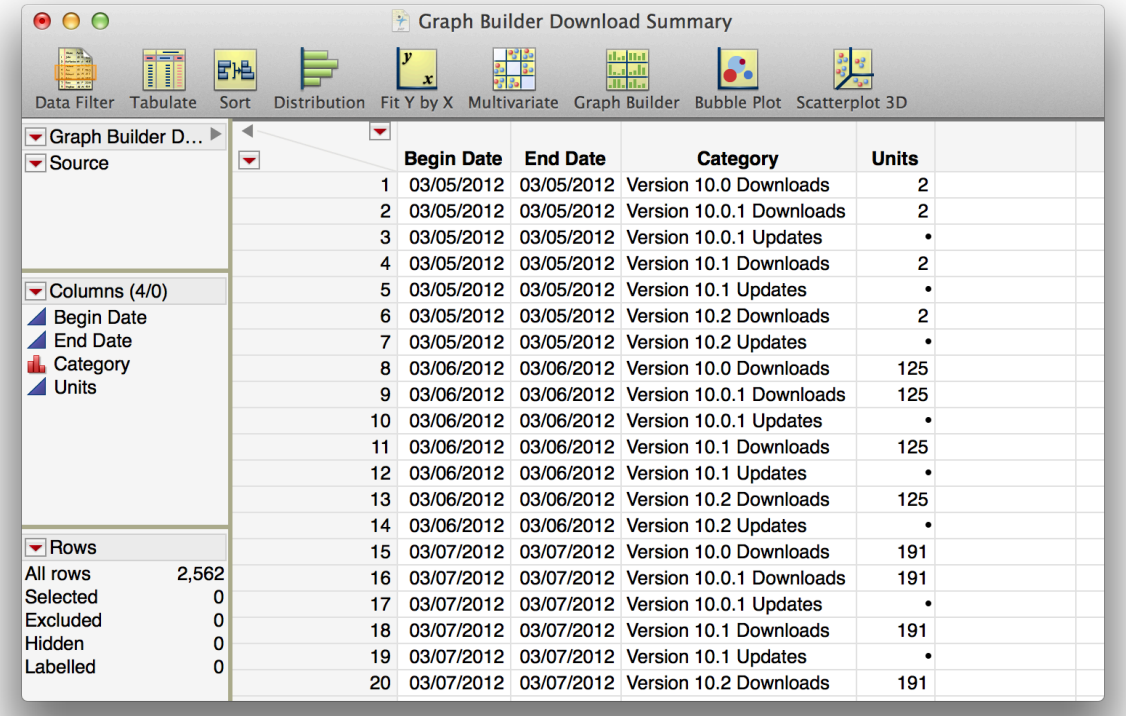
After that, we can add a final element to *sum col strs* for the **Lag()** of the current column. Then we build *sum arg str* by using the **Concat Items()** function to concatenate all our *sum col strs* columns together, separated by commas.

```
download cols = {};

// Add cumulative columns
For( c = 1, c <= N Cols( dt temp ), c++,
	...

	sum col strs = {};
	For( i = 1, i <= N Items( sum cols ), i++,
		Insert Into( sum col strs, Eval Insert( ":Name( \!"^sum cols[ i ] <<
Get Name^\!" )" ));
	);
	Insert Into( sum col strs, Eval Insert( "Lag( :^form col name^, 1 )" ));
	sum arg str = Concat Items( sum col strs, ", " );

	Write( Eval Insert( "\!N^c^: ^form col name^: ^sum arg str^" ));
);
```

Once again, we check the JMP Log to make sure our JSL is doing what we want.

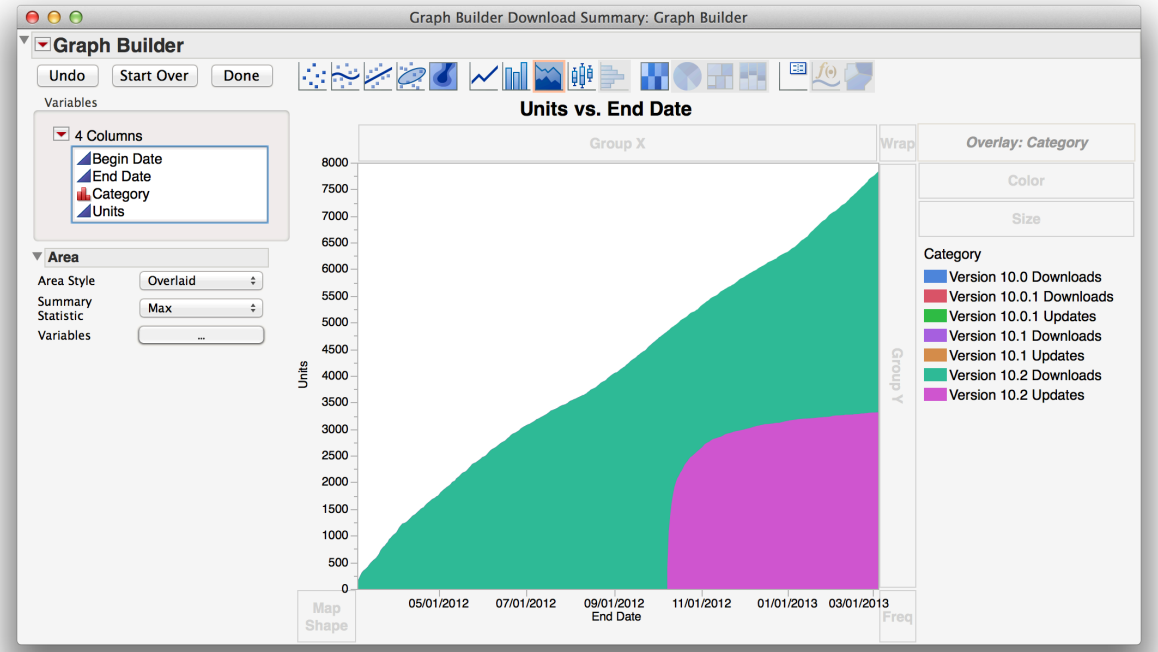
```
4: Version 10.0 Downloads: :Name( "Sum(Units, 10.0, 1T)" ), Lag( :Version 10.0
Downloads, 1 )
5: Version 10.0.1 Downloads: :Name( "Sum(Units, 10.0, 1T)" ), :Name( "Sum(Units,
10.0.1, 1T)" ), Lag( :Version 10.0.1 Downloads, 1 )
6: Version 10.0.1 Updates: :Name( "Sum(Units, 10.0.1, 7T)" ), Lag( :Version 10.0.1
Updates, 1 )
7: Version 10.1 Downloads: :Name( "Sum(Units, 10.0, 1T)" ), :Name( "Sum(Units,
10.0.1, 1T)" ), :Name( "Sum(Units, 10.1, 1T)" ), Lag( :Version 10.1 Downloads, 1 )
8: Version 10.1 Updates: :Name( "Sum(Units, 10.1, 7T)" ), Lag( :Version 10.1
Updates, 1 )
9: Version 10.2 Downloads: :Name( "Sum(Units, 10.0, 1T)" ), :Name( "Sum(Units,
10.0.1, 1T)" ), :Name( "Sum(Units, 10.1, 1T)" ), :Name( "Sum(Units, 10.2, 1T)" ),
Lag( :Version 10.2 Downloads, 1 )
10: Version 10.2 Updates: :Name( "Sum(Units, 10.2, 7T)" ), Lag( :Version 10.2
Updates, 1 )
```

This looks good, so we replace our **Write()** with the **Eval( Parse())** from the previous script, then add back the **Stack()** from Step 4. Now our output data table looks like this.




| | Begin Date | End Date | Category | Units |
|---|---|---|---|---|
| 1 | 03/05/2012 | 03/05/2012 | Version 10.0 Downloads | 2 |
| 2 | 03/05/2012 | 03/05/2012 | Version 10.0.1 Downloads | 2 |
| 3 | 03/05/2012 | 03/05/2012 | Version 10.0.1 Updates | • |
| 4 | 03/05/2012 | 03/05/2012 | Version 10.1 Downloads | 2 |
| 5 | 03/05/2012 | 03/05/2012 | Version 10.1 Updates | • |
| 6 | 03/05/2012 | 03/05/2012 | Version 10.2 Downloads | 2 |
| 7 | 03/05/2012 | 03/05/2012 | Version 10.2 Updates | • |
| 8 | 03/06/2012 | 03/06/2012 | Version 10.0 Downloads | 125 |
| 9 | 03/06/2012 | 03/06/2012 | Version 10.0.1 Downloads | 125 |
| 10 | 03/06/2012 | 03/06/2012 | Version 10.0.1 Updates | • |
| 11 | 03/06/2012 | 03/06/2012 | Version 10.1 Downloads | 125 |
| 12 | 03/06/2012 | 03/06/2012 | Version 10.1 Updates | • |
| 13 | 03/06/2012 | 03/06/2012 | Version 10.2 Downloads | 125 |
| 14 | 03/06/2012 | 03/06/2012 | Version 10.2 Updates | • |
| 15 | 03/07/2012 | 03/07/2012 | Version 10.0 Downloads | 191 |
| 16 | 03/07/2012 | 03/07/2012 | Version 10.0.1 Downloads | 191 |
| 17 | 03/07/2012 | 03/07/2012 | Version 10.0.1 Updates | • |
| 18 | 03/07/2012 | 03/07/2012 | Version 10.1 Downloads | 191 |
| 19 | 03/07/2012 | 03/07/2012 | Version 10.1 Updates | • |
| 20 | 03/07/2012 | 03/07/2012 | Version 10.2 Downloads | 191 |

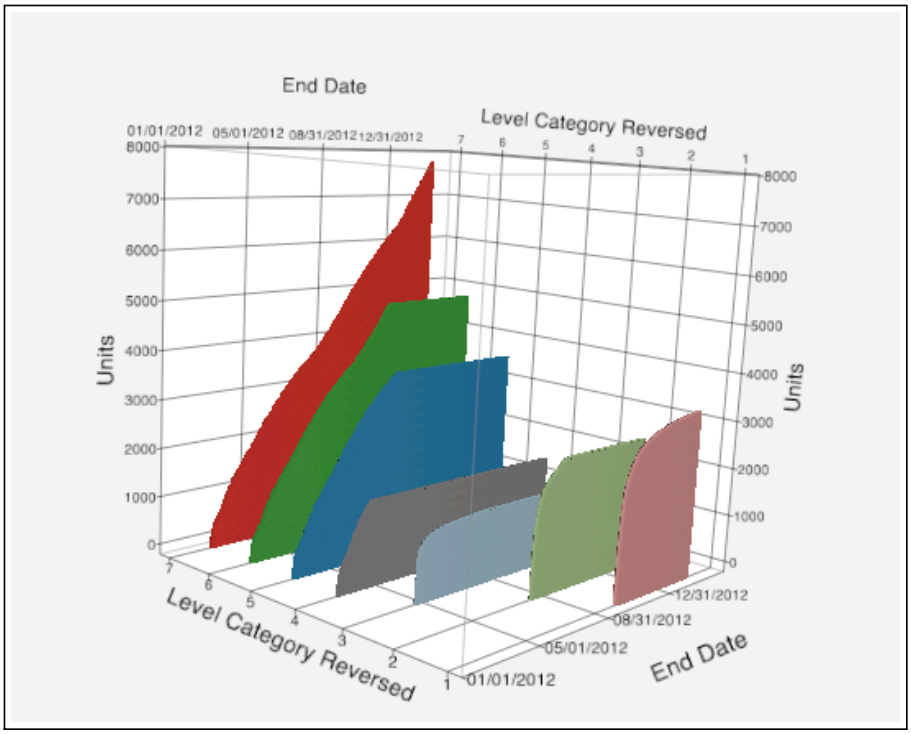
We can actually feed this table directly into *Graph Builder*, like so.




We created an Area graph of *Units* by *End Date*, using *Category* as an overlay variable. We set Area Style to Overlaid, and Summary Statistic to Max. But there's a rather glaring visual problem: most of the data is obscured by *Version 10.2 Downloads*.

## Step 6: Value Ordering

If we look again at our 3-D view of the desired graph, we see that the categories need to be in a specific front-to-back order.




*Version 10.0 Downloads* (#4 above) needs to be sort-of in the "center". Other download categories go behind it in decreasing order to give them a stacked appearance. But update categories go in front of it, in increasing order. Altogether, we would like to give the *Category* column a Value Ordering property that looks like this.

```
// Set Value Ordering column property
Column( dt summary, "Category" )
	<< Set Property( "Value Ordering", {
		"Version 10.2 Downloads",
		"Version 10.1 Downloads",
		"Version 10.0.1 Downloads",
		"Version 10.0 Downloads",		// The middle value
		"Version 10.0.1 Updates",
		"Version 10.1 Updates",
		"Version 10.2 Updates"
	});
```

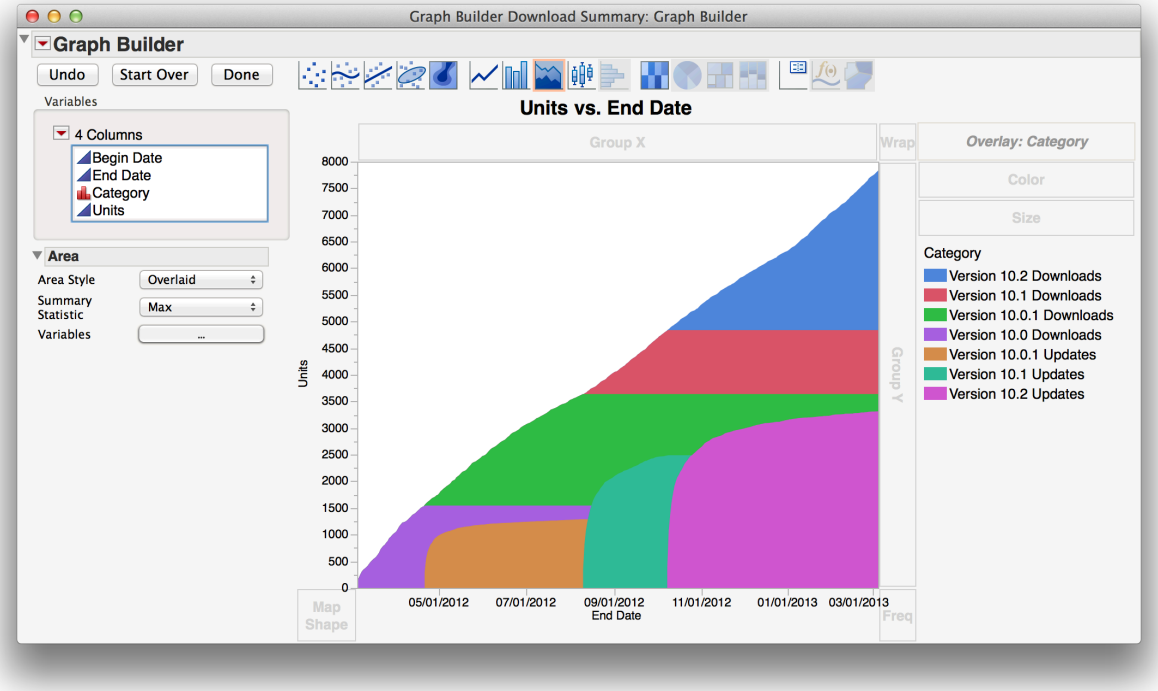
At this point, we should have a good feel for how we can build this dynamically. We need a list of string values, and we can collect them at the same time that we build column formulas. Download columns get *prepended* to (added to the front of) the list, and update columns get *appended* to (added to the end of) the list. If we fold this logic into our existing script, here's how it looks:

```
download cols = {};
stack cols = {};
value order = {};

// Add cumulative columns
For( c = 1, c <= N Cols( dt temp ), c++,
	...

	Match( action,
	"Downloads",
		...
		Insert Into( value order, form col name, 1 );  // prepend
	,
	"Updates",
		...
		Insert Into( value order, form col name );     // append
	);

	...
);

Write( Eval Insert( "\!NValue Order: ^value order^" ));

// Stack the cumulative Units; create the Categories
...

// Add Value Ordering to Category
Column( dt summary, "Category" ) << Set Property( "Value Ordering", value order );
```

We're building another list named *value order*. It starts out empty; then in our column loop, we insert the *form col name* of the column we're currently processing. For Download columns, we prepend it to the list by using the optional third argument to **Insert Into()**. This adds it before position 1, meaning at the beginning of the list. Update columns also use **Insert Into()**, but this time without the third argument. Since the position is not specified, the new item is appended to the end of the list. We check the JMP Log to verify our logic.

```
Value Order: {"Version 10.2 Downloads", "Version 10.1 Downloads", "Version 10.0.1
Downloads", "Version 10.0 Downloads", "Version 10.0.1 Updates", "Version 10.1
Updates", "Version 10.2 Updates"}
```

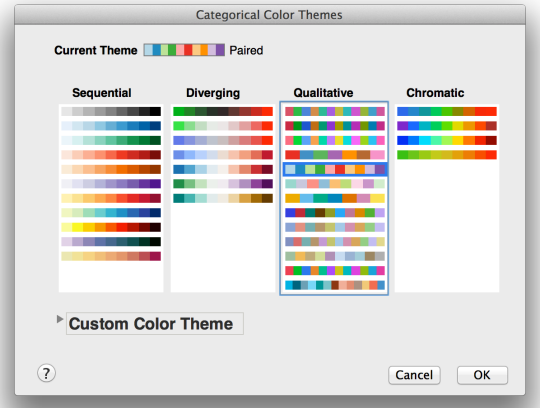
That's exactly the list we want! So after we create the *dt summary* table, we can set the Value Ordering property to *value order* as shown. Now our graph looks almost correct! We just need better colors.




## Step 7: Value Colors

JMP's default color theme doesn't help us match up the downloads for a particular version with their corresponding updates. What we need is a color scheme that works in pairs. And in keeping with our goal of having the script automatically continue to work when new versions are introduced, we don't want to hard-code the colors.

Fortunately, JMP already provides a color theme that will work well for this purpose. It is the categorical color theme aptly named "Paired". This color theme provides pairs of highly contrasting colors, each with a light and a bold variant. Perfect!




What we want to do is assign colors from this theme to specific categories using the Value Colors column property. For each version, we'll give the lighter color of the pair to the "Updates" category and the bolder color to the "Downloads" category. The very first version is unpaired because it only has downloads, so we must treat it as a special case.

The Value Colors property we want to construct needs to look like this:

```
// Set Value Colors column property
Column( dt summary, "Category" )
	<< Set Property( "Value Colors", {
		"Version 10.0 Downloads" = -8421504,
		"Version 10.0.1 Downloads" = -2062516,
		"Version 10.0.1 Updates" = -10931939,
		"Version 10.1 Downloads" = -3383340,
		"Version 10.1 Updates" = -11722634,
		"Version 10.2 Downloads" = -14883356,
		"Version 10.2 Updates" = -16489113
	});
```

This is obviously a list, but a list of what? It appears to be a list of assignment expressions of the form *"category = color",* where *category* is the category as a literal string, and *color* is specified as some sort of numeric value. Obviously, JMP is not assigning a number to a literal string; it's using this list to define the mapping of column values to colors. We need to devise a way to programmatically build this list of expressions!

First, let's tackle the colors. We can use the **Level Color()** function to extract a single color from a color theme. For example, **Level Color(** 1, "Paired" **)** returns -10931939, the first color from the Paired color theme, using the numeric representation we see above. For the special case of the very first version, we can use the **RGB Color()** function to construct a neutral gray like this: **RGB Color( 0.5, 0.5, 0.5 )**. This returns -8421504, the numeric representation for 50% gray.

We can express this in JSL as follows:

```
first = 1;
pair = 1;

// Add cumulative columns
For( c = 1, c <= N Cols( dt temp ), c++,
	...

	Match( action,
	"Downloads",
		// Choose color for this category
		If( first,
			color = RGB Color( 0.5, 0.5, 0.5 );
			first = 0;
		,
			color = Level Color( pair + 1, "Paired" );
		);
	,
	"Updates",
		// Choose color for this category
		color = Level Color( pair, "Paired" );
		pair += 2;
	);

	Write( Eval Insert( "\!N^c^: ^form col name^ = ^color^" ));
);
```

For each column, we set the variable *color* to an appropriate color. We use the variable *pair* to walk through the color theme levels two at a time. For download versions, we use the bolder second color of the pair (*pair* + 1); for update versions we use the first color of the pair and then bump the variable to the next pair. To detect the special case of the first color, we use the variable *first*. When run, we see this output on the JMP Log.

```
4: Version 10.0 Downloads = -8421504
5: Version 10.0.1 Downloads = -2062516
6: Version 10.0.1 Updates = -10931939
7: Version 10.1 Downloads = -3383340
8: Version 10.1 Updates = -11722634
9: Version 10.2 Downloads = -14883356
10: Version 10.2 Updates = -16489113
```

Part one accomplished!

The second part is more challenging: how do we build that list of assignment expressions? We can't simply insert an expression into a list: **Insert Into(** *value colors*, *form col name* = *color* **)**. This just gives us a list of numeric color values, since it evaluates the second argument and assigns *color* to *form col name*. We could try using the **Expr()** function to suppress evaluation of the second argument: **Insert Into(** *value colors*, **Expr(** *form col name* = *color* **))**. But that doesn't work either; we just get a list of identical expressions of *form col name* = *color*. We need a way to evaluate part of the expression, but not the whole thing.

My favorite solution to this conundrum is to use **Eval Expr( Expr())**. You can think of it this way: **Eval Expr()** does the same thing as **Expr()** — it suppresses the evaluation of its argument, returning that argument as an unevaluated expression. But first, it examines its argument looking for **Expr()** functions within it. It does evaluate each of those expressions, then replaces the **Expr()** function with the result. You can think of it as doing the same job as **Eval Insert()**, except that **Eval Insert()** operates on strings and uses the ^ symbol to delimit expressions; **Eval Expr()** operates on expressions and uses the **Expr()** function to delimit expressions.

> Note: In this particular instance, we want to use the expression returned from **Eval Expr()** directly. But more often, we instead want to evaluate that expression and use its result. In that more typical case, we use **Eval( Eval Expr( Expr()))**, which I find particularly mnemonic!

We can put **Eval Expr( Expr())** to work for us like so:

**Insert Into(** *value colors*, **Eval Expr( Expr(** *form col name* **)** = **Expr(** *color* **)))**

If we add this to our code above, then write the result to the JMP Log after the loop, we see this:

```
value colors = {"Version 10.0 Downloads" = -8421504, "Version 10.0.1 Downloads" =
-2062516, "Version 10.0.1 Updates" = -10931939, "Version 10.1 Downloads" = -3383340,
"Version 10.1 Updates" = -11722634, "Version 10.2 Downloads" = -14883356, "Version
10.2 Updates" = -16489113}
```

A list of assignment expressions — perfect! Now we just use **<< Set Property** to assign our value colors to the *Category* column. In fact, we can combine it with the other **<< Set Property** for Value Ordering this way:

```
// Add properties to Category
Column( dt summary, "Category" )
	<< Set Property( "Value Ordering", value order )
	<< Set Property( "Value Colors", value colors );
```

## Final Cleanup

We are 99% there — we just need to tweak a few more items in our script. As before, we delete the **Source** property from *dt summary*, name it, save it, and make a new data view for it.

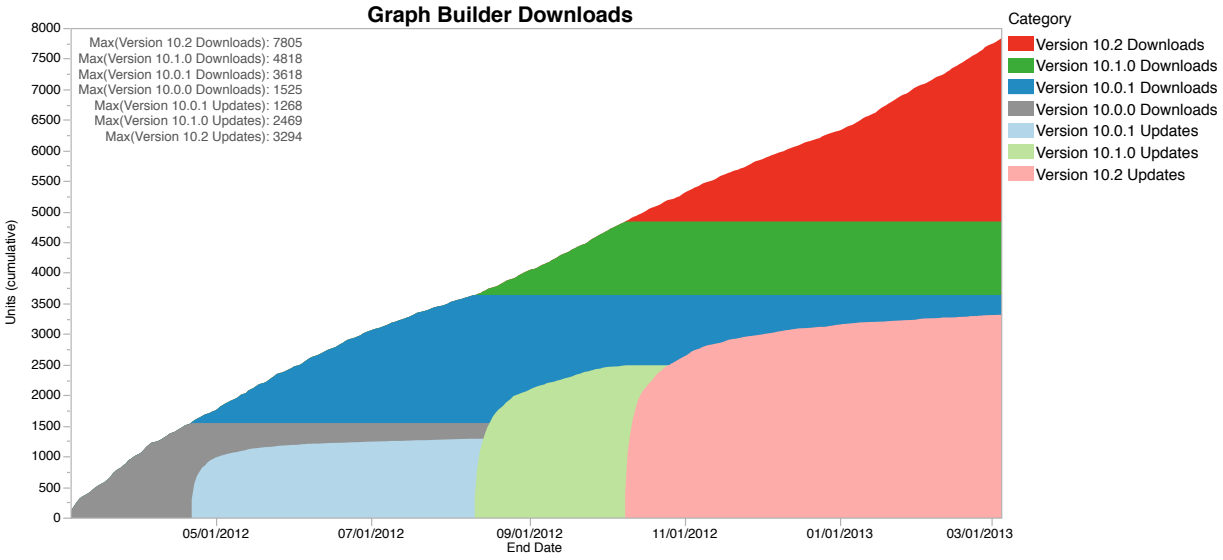
But before saving it, we will add a table script that creates our Graph Builder graph. That is, after all, what this is all about! Our Graph Builder script looks like this:

```
dt summary << New Table Script(
	"Graph Builder",
	Graph Builder(
		Size( 830, 468 ),
		Show Control Panel( 0 ),
		Variables( X( :End Date ), Y( :Units ), Overlay( :Category ) ),
		Elements(
			Area(
				X,
				Y,
				Legend( 3 ),
				Area Style( "Overlaid" ),
				Summary Statistic( "Max" )
			),
			Caption Box(
				X,
				Y,
				Legend( 4 ),
				X Position( "Left" ),
				Summary Statistic( "Max" ),
				Y Position( "Top" )
			)
		),
		SendToReport(
			Dispatch( {}, "400", LegendBox, {Position( {1, 0, 2} )} ),
			Dispatch(
				{},
				"graph title",
				TextEditBox,
				{Set Text( "Graph Builder Downloads" )}
			)
		)
	)
);
```

Then we can tell JMP to run it for us with this command:

```
dt summary << Run Script( "Graph Builder" );
```

And we are rewarded with our beautiful graph, constructed automatically from raw data files, all with JSL.




# Conclusion

Our script uses several techniques to conquer a particular challenge of JSL: substituting a variable for a function parameter. Many times we can just use our variable directly, like we did for the **Columns()** argument to **<< Stack**. This is simple, direct, and efficient. If it's possible to do, this should always be our first choice.

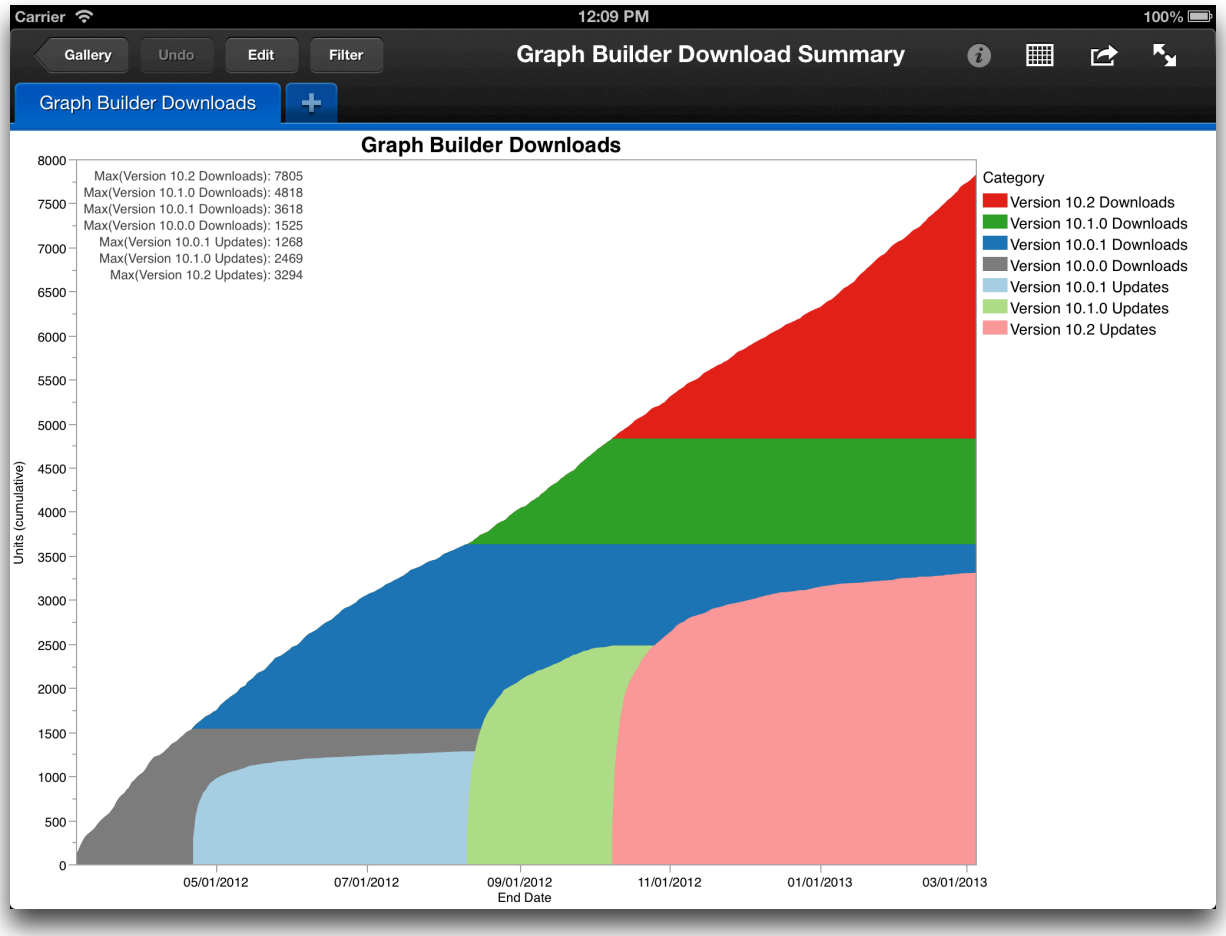
But it's not always possible to do that in JSL. In constructing the Value Colors property, we employed **Eval Expr( Expr())**. This is useful when the argument is an expression that we can't write directly. This technique is not quite as simple and a bit indirect. But it is still very efficient. This should be our second choice, when we can't specify our variable directly.

But when we can't even use **Eval Expr( Expr())**, we can always build a JSL snippet as a string, then use **Eval( Parse())** to parse and run it. We did this to construct our column formulas. This method is cumbersome because we have to deal with issues like embedded double quotes. And it is inefficient because we must **Parse()** the string each time we want to run it. It's much slower to parse a string into an expression each time; it's much better to have the expression already built before your script even runs. But sometimes **Eval( Parse())** is our only option. We should only pull out the "big guns" when our first two options are not feasible.

A useful tool when building that JSL snippet is **Eval Insert()**. This handy string construction function has lots of uses. Using JMP interactively, then scavenging the **Source** property gives us a great starting point for coding many operations. And the useful *Invisible* option makes our JSL

run much faster by not showing intermediate windows. These are all worthy addition to our tool chest.

And finally, to close the loop on this topic, we can transfer our resulting data table back to the iPad, where we can display it in the very app whose download data we have been analyzing!




# Contact information

Your comments and questions are valued and encouraged. Contact the author at:


Michael Hecht
michael.hecht@jmp.com




# References


[1] JMP Graph Builder for iPad. http://www.jmp.com/software/jmp10/jmp-graph-builder-for-ipad.shtml

[2] Apple Autoingestion tool. http://www.apple.com/itunesnews/docs/Autoingestion.class.zip