# Most Common JSL Mistakes and How to Avoid Them

JMP Discovery Summit Americas 2020

Wendy Murphrey – Sr. Principal Technical Support Analyst, SAS
Justin Chilton – Sr. Associate Software Development Engineer in Test, SAS

In version 4, a scripting language known as JMP Scripting Language, or JSL, was introduced as a useful new feature of JMP. Celebrated for its easy point-and-click interface, JMP users had longed for a way to repeat their interactive steps to streamline the analysis of fresh data. JSL was the answer. Over time, JSL has grown to be an integral part of how many JMP users make important discoveries in their data.

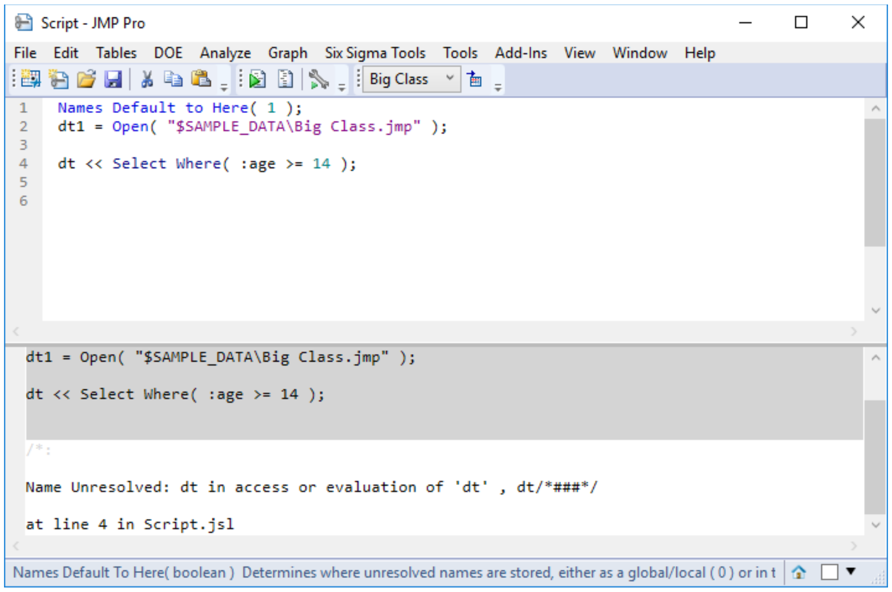
In this paper, we will discuss several common scenarios that script authors of various experience levels encounter. Our intent is to explain what is happening so that you have the tools and knowledge to accomplish your scripting goals.

## Mistake: Not Checking the Log

Perhaps because it is not open by default, it is easy to forget to check the log for messages. Some ways to access the log:

- Select View > Log (on Windows operating systems) or Window > Log (on Macintosh operating systems).
- Right-click inside the script window and select **Show Embedded Log** to see the log inside your script window.

Here you will find useful information about recent script executions. For example, error messages, row numbers, and script file names are displayed to help guide you to where trouble might be lurking in your script. Notice in the following example, the script author simply used the wrong data table reference for the Select Where(). The embedded log identifies the error is on line 4 and provides the `Name Unresolved` error message which indicates that JMP was not able to resolve the reference or name. The error was generated in this instance because the data table reference is dt1, not dt.


Script - JMP Pro
File Edit Tables DOE Analyze Graph Six Sigma Tools Tools Add-Ins View Window Help
Big Class

```
Names Default to Here( 1 );
dt1 = Open( "$SAMPLE_DATA\Big Class.jmp" );

dt << Select Where( :age >= 14 );
```

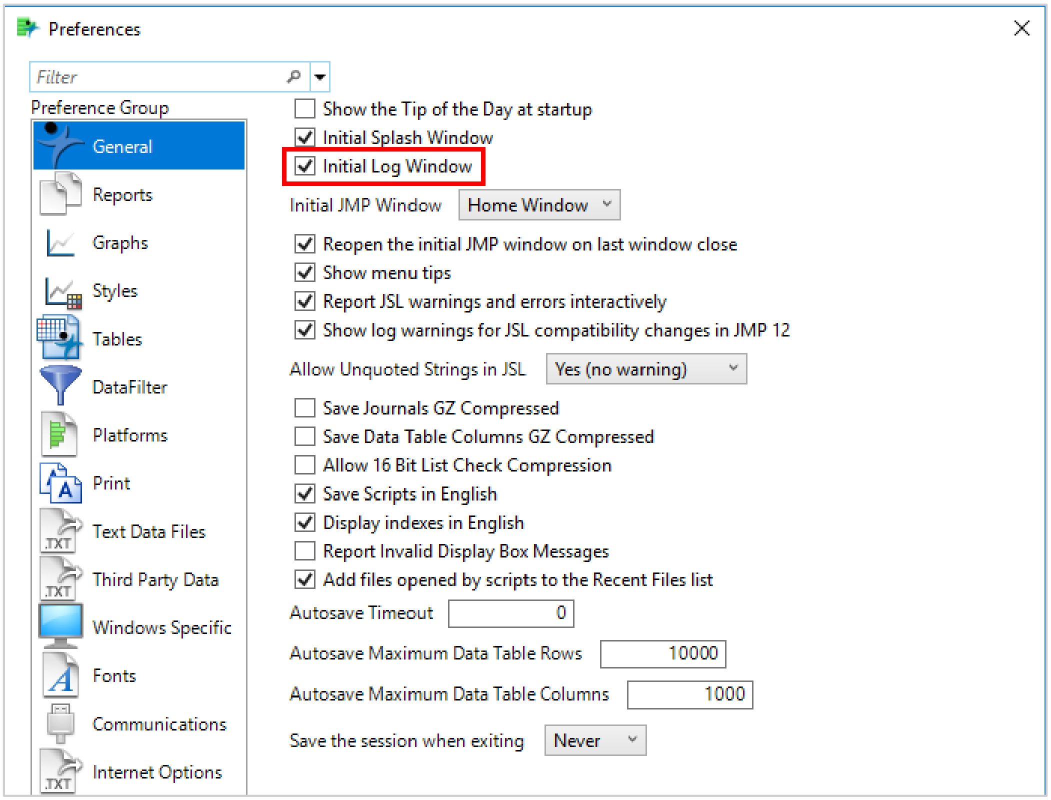
```
dt1 = Open( "$SAMPLE_DATA\Big Class.jmp" );

dt << Select Where( :age >= 14 );

/*:

Name Unresolved: dt in access or evaluation of 'dt' , dt/*###*/

at line 4 in Script.jsl
```

Names Default To Here( boolean ) Determines where unresolved names are stored, either as a global/local ( 0 ) or in t

So the takeaway is to always check the log – especially when what happens does not meet your expectation. It might be as simple as a misspelled name.

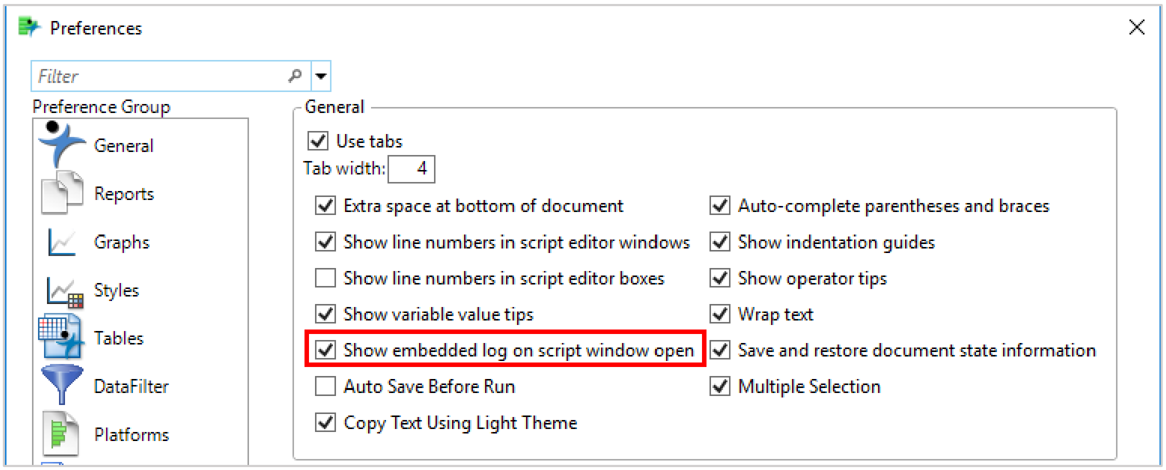
Keep in mind that there are preferences to specify that the log window is always available:

- With the General preference group selected, enable the check box beside **Initial Log Window** (Windows only).



☐ Show the Tip of the Day at startup
☑ Initial Splash Window
☑ Initial Log Window
Initial JMP Window: Home Window
☑ Reopen the initial JMP window on last window close
☑ Show menu tips
☑ Report JSL warnings and errors interactively
☑ Show log warnings for JSL compatibility changes in JMP 12
Allow Unquoted Strings in JSL: Yes (no warning)
☐ Save Journals GZ Compressed
☐ Save Data Table Columns GZ Compressed
☐ Allow 16 Bit List Check Compression
☑ Save Scripts in English
☑ Display indexes in English
☐ Report Invalid Display Box Messages
☑ Add files opened by scripts to the Recent Files list
Autosave Timeout: 0
Autosave Maximum Data Table Rows: 10000
Autosave Maximum Data Table Columns: 1000
Save the session when exiting: Never

- With the Script Editor preference group selected, enable the check box beside **Show embedded log on script window open**.



General
☑ Use tabs
Tab width: 4
☑ Extra space at bottom of document
☑ Auto-complete parentheses and braces
☑ Show line numbers in script editor windows
☑ Show indentation guides
☐ Show line numbers in script editor boxes
☑ Show operator tips
☑ Show variable value tips
☑ Wrap text
☑ Show embedded log on script window open
☑ Save and restore document state information
☐ Auto Save Before Run
☑ Multiple Selection
☑ Copy Text Using Light Theme

## Mistake: Depending on the Current Data Table

The concept of Current Data Table is often described as the active table. But what makes a table the active (or current) data table? And why does that matter to your script?

When a table is opened, created, or otherwise acted upon, it becomes the current data table. But there are other ways that a table can become current that you, as the script author, might not have anticipated.

There are some tasks that JMP processes in the background. For example, JMP may need to evaluate a formula or make an update to a window. It is these background processes that can cause a different table than the one you intended to become the active, or current, data table at runtime. Because it is not the intended table, this can cause your script to fail – or worse, you don't realize that JMP used the wrong table.

The best practice is to craft your scripts using data table specific references to ensure that JMP uses the table you intend regardless of what table is current at runtime. Below are some examples of how to guard your scripts against depending upon the active or current table.

- Send platform calls as messages to the desired data table reference:

  ```
  dt1 << Bivariate( Y( :weight ), X( :height ), "Fit Line" );
  ```

- Create new columns using New Column() as a message instead of a function:

  ```
  dt1 << New Column( "Mean Height by Age",
    "Numeric",
    Formula( Col Mean( :height, :age ) )
  );
  ```

- Use optional data table references when referencing columns:

  ```
  Column( dt1, "age" )
  //or
  dt1:age
  //or
  As Column( dt1, "age" )
  ```

- Use optional data table references in functions, where available:

  ```
  For Each Row( dt1, <script> );
  Summarize( dt1, <options> );
  ```

- Avoid assigning a data table reference using the Current Data Table() function:

  ```
  dt << Join(
        With( dt2 ),
        Select( :popcorn, :oil amt, :batch, :yield ),
        SelectWith( :yield ),
        By Matching Columns( :popcorn = :popcorn, :batch = :batch )
  );
  newDT = Current Data Table(); //Not recommended
  ```

  Because Join() returns a reference to the newly created table, assign the reference when the Join() is called and your reference will represent the newly Joined table regardless of what table is current.

```
/* Solution */
newDT = dt << Join(
      With( dt2 ),
      Select( :popcorn, :oil amt, :batch, :yield ),
      SelectWith( :yield ),
      By Matching Columns( :popcorn = :popcorn, :batch = :batch, :oil amt = :oil ),
      Output Table Name( "My Joined Table" )
);
```

## Mistake: Mis-using Column() In Select Where

Being able to identify a column using a JSL variable is a common task and there are several ways to do it. So how do you know which one to use? It all depends upon the intended use. For example, a platform can consume a reference to the column as a whole. Meanwhile, a formula evaluates on every row and would need a reference to the value on each row of the column.

The Column() function returns a reference to the column as a whole unit. The function is flexible for a couple of reasons. A variety of arguments are accepted: column name as a string, a string variable, an index, or a numeric variable representing an index. It can also be subscripted to reference the column's value on a row, as in `Column( "age" )[5]`, or the current row, as in `Column( "age" )[Row()]`.

Some places that you can use the Column() function are in platform calls, sending messages to a column, or assigning a variable reference to a column.

The As Column() function returns a reference to the value on the current row of the specified column. It accepts the same arguments as the Column() function. The As Column() function operates in the same way as the following:

```
:age
dt:age
```

That is, running these lines on their own will always return a missing value. Why? In JMP, the current row is always zero unless otherwise managed. So in the case of a formula or a For Each Row() loop, the row number is controlled by JMP during the evaluation.

Let's look at an example with Select Where. Here we are opening a table and selecting rows that meet a condition.

```
Names Default To Here( 1 );
dt = Open( "$SAMPLE_DATA\Big Class.jmp" );

colName = "age";
dt << Select Where( Column( colName ) == 14 );
```

Because Select Where evaluates on each row, using the Column() function without a row subscript will not select any rows. Why? Remember the Column() function returns a reference to the column as a whole unit. So a whole column of data will never equal 14.

The solutions are to include a Row() subscript, which means the current row, or you can use As Column().

```
dt << Select Where( Column( colName )[Row()] == 14 );
// Or
dt << Select Where( As Column( colName ) == 14 );
```

## Mistake: Leaving Variables in Column Formulas

The use of JSL variables in formulas can be problematic for two primary reasons:

- The lifespan of a JSL variable is limited to the current JMP session.
- As the value of the variable changes, so will the result of the formula.

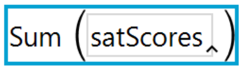
Unless you look at the formula in the formula editor, you may not even realize there is a problem.

So let's look at an example of using a JSL variable in a formula:

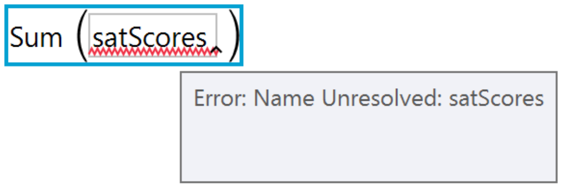
```
dt = Open( "$SAMPLE_DATA\SATByYear.jmp" );

satScores = {:SAT Verbal, :SAT Math};

dt << New Column( "SAT Composite",
        Numeric,
        Continuous,
        Formula( Sum( satScores ) )
);
```

In this example, I am opening a sample data table and establishing a list of variables that I would like to sum in a new column. I have a New Column() statement and I have used the variable for summing in a formula.

When we run this, notice that the variable is showing in the formula.


Sum ( satScores )

If you save the table and close this JMP session, the variable satScores will no longer have a definition in the new JMP session. Looking in the formula editor, you will see a Name Unresolved error.


Sum ( satScores )

Error: Name Unresolved: satScores

When we re-evaluate the formula and select Ignore Errors, all the data will be changed to missing. How do we fix it?

When you use a variable in a formula, you will need to do something to instruct JMP to replace the variable with its value before evaluating the formula. This is where Eval Expr() comes in.

```
//Evaluate the entire New Column expression AFTER EvalExpr() is finished
Eval(
        Eval Expr(
                dt << New Column( "SAT Composite",
                        Numeric,
                        Continuous,
                        Formula( Sum( Expr( satScores ) ) )
                )
        )
);
```

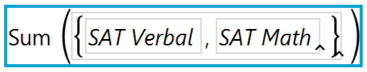
According to the documentation, Eval Expr() evaluates only the inner expressions and returns an expression that contains the results of the evaluation.

What does that mean? Essentially, Eval Expr() tells JMP to evaluate/replace any variables in the argument that are wrapped in an Expr() function and return the UNEVALUATED expression.

So in the above example script, JMP will replace the Expr( satScores ) with {:SAT Verbal, :SAT Match}. Let’s run just the Eval Expr() and check the result in the log to confirm.

```
Eval Expr(
	dt << New Column( "SAT Composite",
		Numeric,
		Continuous,
		Formula( Sum( Expr( satScores ) ) )
	)
)
/*:

dt << New Column( "SAT Composite",
	Numeric,
	Continuous,
	Formula( Sum( {:SAT Verbal, :SAT Math} ) )
)
```

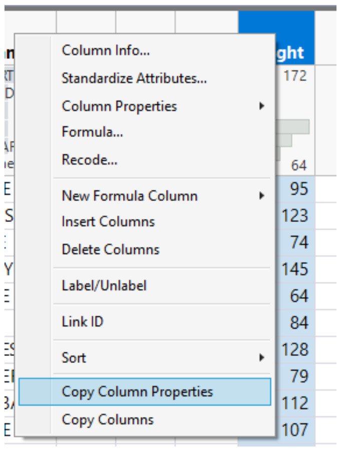
The outer Eval() will cause JMP to evaluate the entire New Column statement after the replacement has occurred. Running the code with Eval(), we can see the correct value in the formula editor.



## Mistake: Incorrect Syntax for Setting Column Properties

Column properties are used to store information related to a specific column. The column is simply a repository for the information and does not verify the syntax. It is the consumer of the column property that is expecting to receive the information in a specific format. This is where the trouble will likely show itself.

To get the proper syntax for a specific column property, add the property interactively first. This will assign the property to the column correctly. You can then capture the script that will set the property using Get Property() to show the script in the log. Optionally, you can right-click the column name and select Copy Column Properties to place the Set Property() script on your clipboard.



After copying the column properties from the context menu, you can paste it in a script editor, and it will look like the script below. There may be multiple properties depending on the source column.

```
Add Column Properties( Set Property( "Notes", "Explore data adventurously" ) )
```

To set just one of the column properties, remove the Add Column Properties section, and send the Set Property message to the desired column.

```
:my column <<Set Property( "Notes", "Explore data adventurously" )
```

This process can be used to interactively get the syntax for any column property.

## Mistake: Manipulating Dispatch Commands

Dispatch commands were initially developed as an internal method for JMP to use to recreate graphs with customizations that were made through the interface. The intent was that scripted changes would be accomplished through messages directed to the appropriate object, such as an AxisBox to update the axis settings.

It can be surprising when attempting to change a Dispatch command and finding out it does not accept user variables. Below are some solutions to this common mistake.

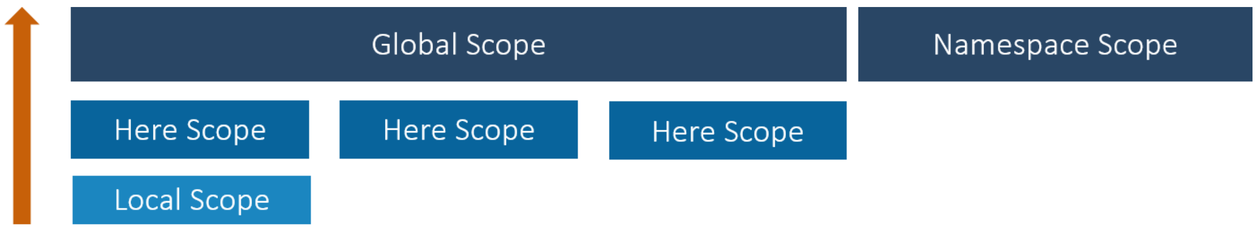
- Use complicated Eval/EvalExpr/Expr solution – not recommended
- Use DisplayBox subscripting or XPath to access box and send it message directly

## Mistake: Only Using the Global Namespace

Only using the global namespace is great for quick scripts or scripts for your own use, but when writing scripts or applications to share with others, it is a good idea to put your symbols in a namespace.

Each script comes with their own Here namespace for free or you can create your own custom namespace. Use these scopes to separate your JSL symbols from the symbols from other scripts.

The graphic below shows the separation between Here scopes and the separation of namespace scopes and everything else. Since the lookup order happens from the bottom up, using this technique ensures symbols are safe from being overwritten by other scripts.



## Mistake: Using Default Local in Functions

Using Default Local to manage a function's local variables can be the easiest way to manage locals but can be quite tricky too.

One key thing to note is that Default Local is a **parse time** directive and not a runtime state (like Names Default to Here). Therefore, JMP assumes what the locals are by looking at your function (mostly for assignments), making it possible for it to incorrectly assume you wanted a local when you in fact wanted a symbol outside of the function. Because of this, without appropriate variable scoping, non-local variables are unable to be manipulated

One problem that can occur is when unscoped variables unexpectedly resolve as locals if they are modified directly within the function.

Below are some solutions to this problem.

- Use explicit locals list
- Explicitly scope non-locals

## Mistake: Mis-using Locals in Functions

When using functions in JSL, it is good to hide any variables not used outside the function.

There are 3 ways to make local variables

- A locals list: {a, b, c}
- Explicit scoping: local:a
- Default Local: {Default Local}

Using locals can be confusing when a function creates display boxes in non-modal windows. Callbacks for DisplayBoxes cannot access the locals from when the box was created because the local variables are already destroyed.

Below are some solutions to this problem.

- Define variables outside of the function
- Inject variable value's into callback function
- Use a modal dialog

## Mistake: Overlooking Resources

There are many resources available to you to learn more about JSL. And many are free!

- Scripting Index (Help > Scripting Index)
- Online Documentation (Help > JMP Help)
- JMP User Community
  - Blogs
  - Discussions

    - File Exchange
- On-Demand Webcasts
    - Application Development
- Technical Support
    - Report a Problem
    - support@jmp.com
- Books
    - Jump into JMP Scripting, Second Ed.
    - JSL Companion: Applications of the JMP Scripting Language, Second Ed.
- Training
- Mentoring Services