# Using JSL to Develop Efficient, Robust Applications

## JMP Discovery - Prague 2017

Joseph Morgan
JMP Division
SAS Institute Incorporated
Cary, North Carolina 27513
Joseph.Morgan@sas.com

# JSL Application Development

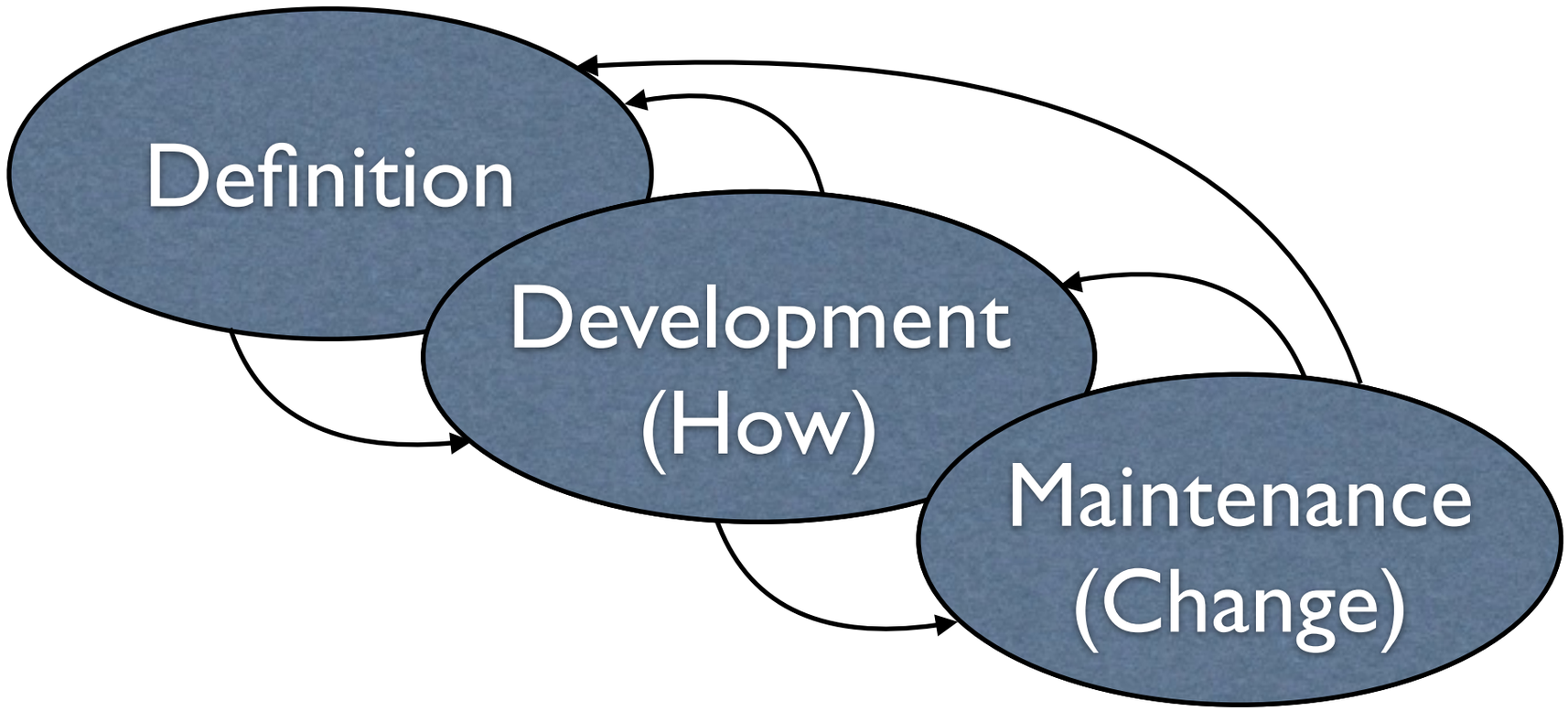
## A Generic view of Software Engineering

In his seminal text "*Software Engineering: A Practitioners Approach*," (2009) Roger Pressman makes the following point:

"The software development process contains three generic phases regardless of the software engineering paradigm chosen. The three phases, ***definition***, ***development***, and ***maintenance***, are encountered in all software development, regardless of application area, project size, or complexity."

# JSL Application Development

## A Generic view of Software Engineering[1]




[1]Pressman, R., *Software Engineering: A Practitioners Approach*, (2009)

# JSL Application Development

*Question*

Are there JSL constructs (or categories of constructs) that programmers should make use of as they tackle the development of non-trivial applications that are efficient, robust, and maintainable?

That is, are there JSL constructs that aid in the **How** phase, yet facilitate the inevitable **Change** phase, of application development.

**Yes!**

# JSL Application Development

**Top 5 JSL constructs.**

1. Allow manipulation of JSL expressions - *The wonders of JSL expression handling functions*.
2. Provide rich, powerful, display tree navigation - *The power of JSL XPath querying*.
3. Guard against name collision - *The security of JSL namespaces*.
4. Provide rich, powerful, matrix algebra capability - *The power of JSL matrices* (the other primitive data type) *and functions*.
5. Provide ordered/unordered general container capabilities - *The remarkable utility of associative arrays and lists*.

# JSL Expressions

## The Wonders of JSL Expression Handling Functions

# JSL Application Development

## Motivating Example:

```
dt = Open( "$SAMPLE_DATA/Socioeconomic.jmp" );
Principal Components(
   Y( 1 :: N Col( dt ) ),
   Estimation Method( "Row-wise" ),
   "on Correlations",
   Factor Analysis( "PC", "SMC", 2, "Varimax" )
);
```

*Challenge*: Write a script that allows “Factoring Method” to be a user specified value. That is, the first argument of `Factor Analysis`(…) may be “PC” or “ML”.

# JSL Demo

The `Principal Components`(…) challenge

# JSL Application Development

## The Wonders of Expression Handling Functions

This section attempts to unravel the mystery surrounding JSL expression handling functions and show how such functions can be used to solve nontrivial JSL programming challenges.

*Question*: What exactly is a JSL expression?

> "*A JSL expression is any combination of variables, constants, and functions linked by operators that can be evaluated*[1]*.*"

The key phrase here is "... *that can be evaluated*."

[1]SAS Institute, Inc., JMP Scripting Guide, Cary, NC: SAS Institute, Inc.

# JSL Application Development

## Preliminaries

Each of the following is a JSL expression.

```
100.1                          // numeric literal - primitive
"string literal"               // string literal - primitive
x                              // variable (or name)
x & (y | z)                    // logical expression
z*2 + z^2 -10 + pi()           // arithmetic expression
```

More complex examples like the following are also JSL expressions.

```
x = [];
for(i=1, i<=5, i++,
   x ||= random uniform(); show(x)
);
```

# JSL Application Development

## Preliminaries

Let us re-examine this expression:

```
x = [];
for(i=1, i<=5, i++,
   x ||= random uniform(); show(x)
);
```

The following script is operationally equivalent:

```
glue(assign(x, []),
   for(assign(i, 1), less or equal(i, 5), post increment(i),
      glue(concat to( x,random uniform()), show(x))
   )
)
```

# JSL Application Development

## What is an Expression Handling function?

A useful way to think of expression handling functions is as the set of JSL functions that enables you to regard expressions as data.

1. **Quoting, Retrieval, and Evaluation:** Functions such as `Expr()`, `NameExpr()`, and `Eval()` allow you to quote expressions, possibly assign them to variables for later retrieval, and for future evaluation.
2. **Assembly and Disassembly:** There are functions that allow expressions to be assembled and disassembled. `Substitute()`, `Insert()`, and `Remove()` are three of several functions that may be used to assemble and disassemble expressions.
3. **Probing:** `Arg()` and `Head()` are functions used to probe expressions.

# JSL Application Development

## How do Expression Handling functions work?

These JSL functions fall into two operational categories: those that evaluate their arguments when invoked and those that do not.

| Evaluate Arguments | Do Not Evaluate Arguments |
|---|---|
| Parse() | Expr() |
| Eval() | NameExpr() |
| EvalList() | EvalExpr() |
| Substitute()/SubstituteInto() | Arg() |
| Remove()/RemoveFrom() | NArg() |
| Insert()/InsertInto() | Head() |
| | HeadName() |

**JSL Expression Handling Functions - Operational Behavior[1]**

[1]Morgan, J., “Expression Handling Functions: Unravelling the Expr(), NameExpr(), Eval(), … Conundrum,” JMPer Cable, #26.

# JSL Demo

## Expression Handling Functions

## A “Deep Dive” (i.e. a detailed examination)

# JSL Application Development

## Pitfall #1a: `Expr()` *vs.* `NameExpr()`

A common JSL mistake is to assume that executing Expr(x) is equivalent to executing NameExpr(x). Indeed, in the following example, these two statements return the same thing.

```
Expr(4 + 35);
NameExpr(4 + 35);

//:*/
Expr(4 + 35);
/*:
4 + 35

//:*/
NameExpr(4 + 35);
/*:
4 + 35
```

Expr(x) returns *its argument unevaluated* but NameExpr(x) returns *the value of its argument unevaluated*. The argument to NameExpr(x) should be a variable, but when it is an expression it simply returns its argument.

# JSL Application Development

## Pitfall #1b: `Expr()` *vs.* `NameExpr()`

Consider:

```
x = Expr(2 + 50);
Expr(x);
NameExpr(x);

//:*/
Expr(x);
/*:
x

//:*/
NameExpr(x);
/*:
2 + 50
```

Since `Expr()` returns its *argument unevaluated*, x is returned, whereas `NameExpr(x)` returns the *value of its argument unevaluated* so **2 + 50** is returned.

**Point**: Executing `Expr(x)` is not equivalent to executing `NameExpr(x)`.

# JSL Application Development

## Pitfall #2: `Eval()`

When using the `Eval()` function, a common mistake is to assume that executing `Eval(x)` is equivalent to executing x. Consider the following example.

```
x = Expr(4 + 25);
x;
x = Expr(4 + 25);
Eval( x );


//:*/
x = Expr(4 + 25);
x;
/*:
29


//:*/
x = Expr(4 + 25);
Eval( x );
/*:
29
```

Statements 1 and 3 store the expression **4 + 25** in x. Execute statements 1 and 2 and then 3 and 4.

**Point**: Executing **Eval( x )** is not equivalent to executing **x**.

# JSL Application Development

Substitute()
evaluates its arguments, it attempts to evaluate stmt, but fails because colx does not exist.

## Pitfall #3: `Substitute()` *vs.* `SubstituteInto()`

The following script uses Substitute() to replace colx, with weight, but fails.

```
//script 1
stmt = Expr(distribution(column(_x_)));
colnm = "weight";
Result = Substitute(stmt, Expr(_x_), colnm);
show(stmt, Result);
```

Properly *quote* the first argument of Substitute(). That is, use NameExpr() to retrieve the value of stmt.

```
//script 1 - revised
stmt = Expr(distribution(column(_x_)));
colnm = "weight";
Result = Substitute(NameExpr(stmt),Expr(_x_),colnm);
show(stmt, Result);
```

# JSL Application Development

## Pitfall #3: `Substitute()` *vs.* `SubstituteInto()`

Alternatively, `SubstituteInto()` may be used.

```
//script 2
stmt = Expr(distribution(column(_x_)));
colnm = "weight";
SubstituteInto(stmt, Expr(_x_), colnm);
show(stmt);
```

`SubstituteInto()` updates the named expression (i.e. stmt) in place.

**Point**: Unlike `Substitute()`, `SubstituteInto()` does not evaluate its first argument.

# JSL Application Development

## Concluding Comments

The primary purpose of this exercise was to illustrate the use of several expression-handling functions. A secondary purpose was to point out common errors, pitfalls, and misunderstandings that JSL programmers sometimes make when attempting to use these functions.

Hopefully those objectives have partly been achieved.

# Display Tree Subscripting

## The Power of JSL XPath querying

# JSL Application Development

## Motivating Example:

```
New Window("Checkbox state",
   Outlinebox("Outline 1", Tablebox(
      Numbercolbox("Number 1",[1,2])), Checkbox({"check 1"})
   ),
   Outlinebox("Outline 2", Tablebox(
      Numbercolbox("Number 2",[1111,2222])), Checkbox({"check 2"})
   ),
   // determine checkbox state
   Buttonbox("Commit",<<setfunction(
      function({t},{l}, l = callback(t); show(l) ) )
   )
);
```

*Challenge*: Implement `callback(t)`. Make sure that your function is general. The number of outline boxes is only known at runtime.

# JSL Demo

## The `Checkbox(...)` state challenge

# JSL Application Development

## The Power of JSL XPath[1] querying

This section attempts to illustrate the power of XPath querying and to show how XPath may be used as an alternative to JMP's existing display tree subscripting construct.

*Question*: What exactly is XPath?

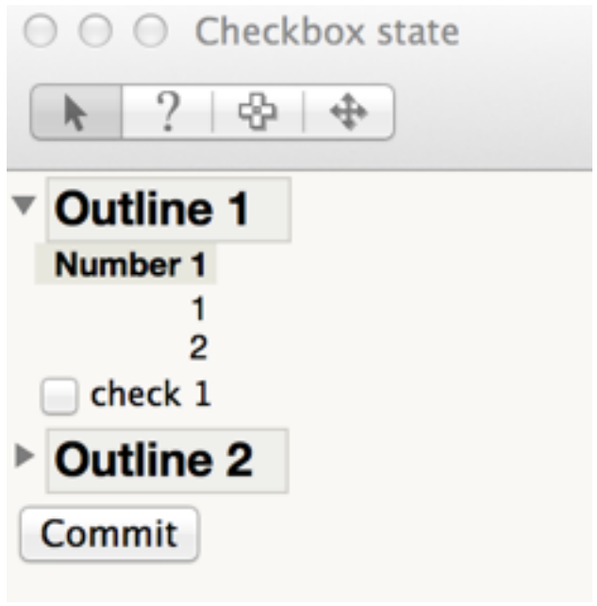
*Answer*: XPath (XML Path Language) is a "World Wide Web Consortium" (W3C) standard query language for addressing parts of an XML document. You may think of XPath as *SQL for XML*.

The key phrase here is "... as *SQL for XML*."

[1]Clark, J., & DeRose, S., XML path language (XPath) v1.0, W3C Recom. (1999), See http://www.w3.org/TR/xpath.html.

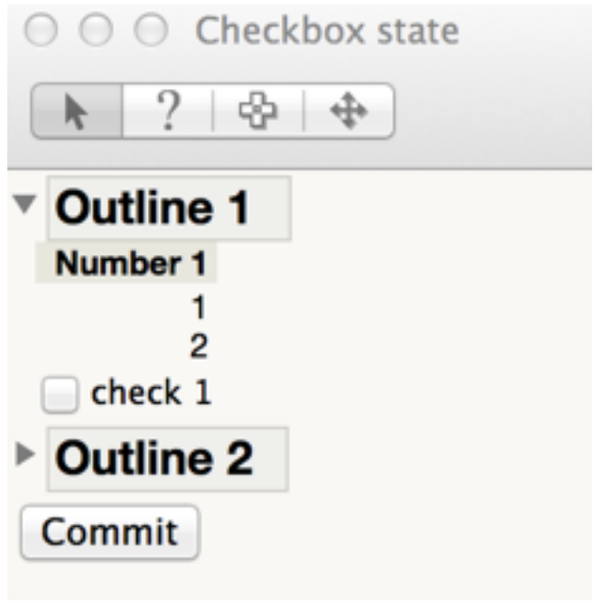
# JSL Application Development

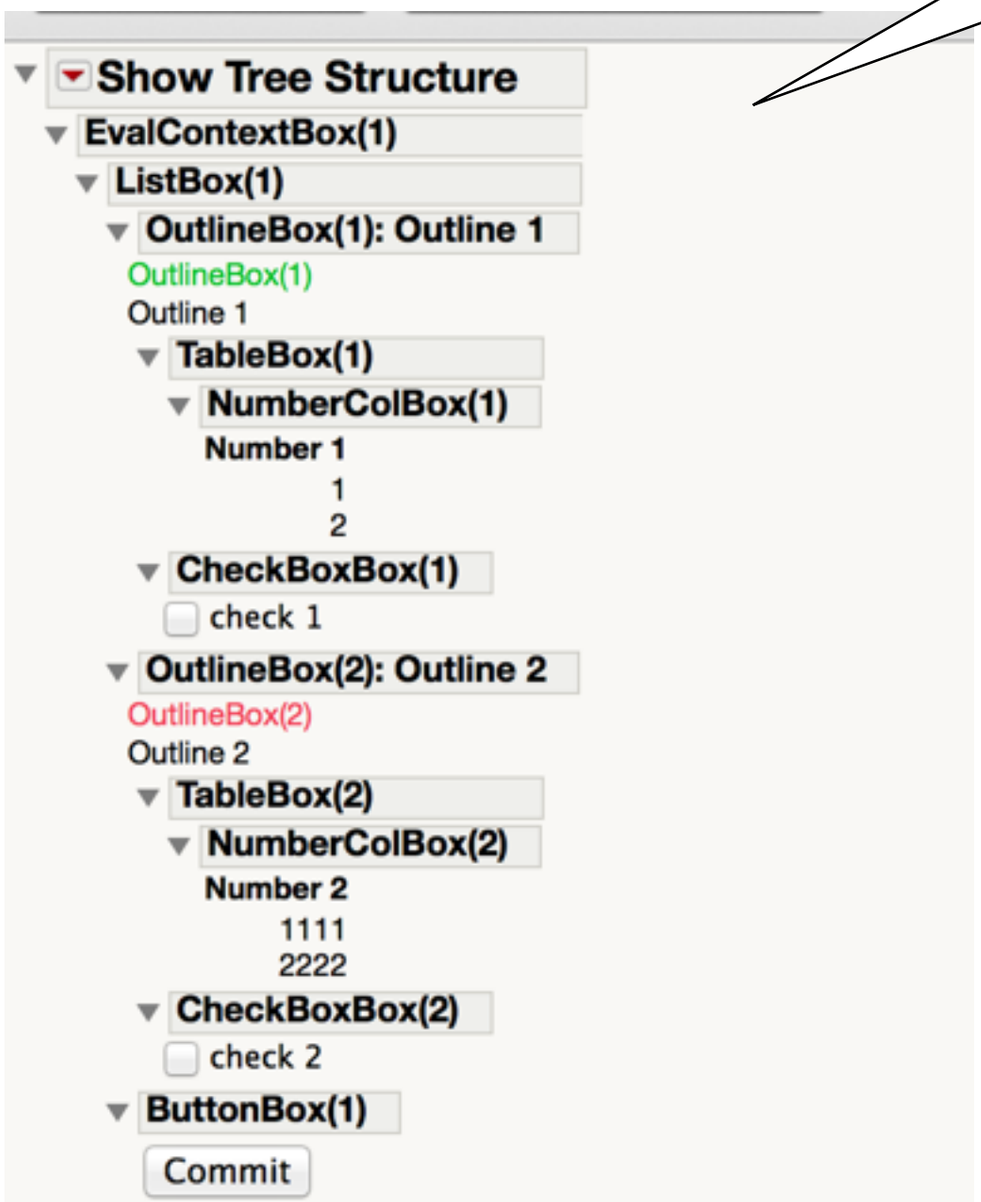
## How does XPath work for display trees?



# JSL Application Development

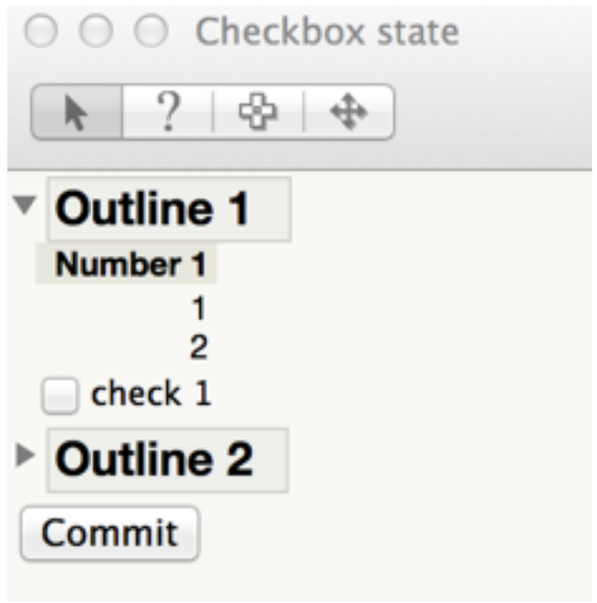
## How does XPath work for display trees?



# JSL Application Development

## How does XPath work for display trees?




**Point**: Given a reference to a JMP report, XPath queries can return text or display box references.

Display Tree -XML

```
<EvalContextBox width=\!"104\!" height=\!"149\!">
  <ListBox width=\!"104\!" height=\!"149\!">
    <OutlineBox width=\!"104\!" height=\!"91\!" isOpen=\!"true\!">Outlin
\!"26\!" width=\!"67\!" height=\!"46\!">
        <NumberColBox width=\!"66\!" height=\!"44\!">
          <NumberColBoxHeader>Number 1</NumberColBoxHeader>
          <NumberColBoxItem>1</NumberColBoxItem>
          <NumberColBoxItem>2</NumberColBoxItem>
        </NumberColBox>
      </TableBox>
      <CheckBoxBox leftOffset=\!"12\!" topOffset=\!"72\!" width=\!"67\!"
        <CheckBoxBoxItem>check 1</CheckBoxBoxItem>
      </CheckBoxBox>
    </OutlineBox>
    <OutlineBox leftOffset=\!"0\!" topOffset=\!"91\!" width=\!"103\!" he
            isOpen=\!"false\!">Outline 2<TableBox leftOffset=\!"12\!" to
        <NumberColBox width=\!"66\!" height=\!"44\!">
          <NumberColBoxHeader>Number 2</NumberColBoxHeader>
          <NumberColBoxItem>1111</NumberColBoxItem>
          <NumberColBoxItem>2222</NumberColBoxItem>
        </NumberColBox>
      </TableBox>
      <CheckBoxBox leftOffset=\!"12\!" topOffset=\!"72\!" width=\!"67\!"
        <CheckBoxBoxItem>check 2</CheckBoxBoxItem>
      </CheckBoxBox>
    </OutlineBox>
    <ButtonBox leftOffset=\!"0\!" topOffset=\!"117\!" width=\!"78\!" hei
```

# JSL Demo

## Xpath Queries

## A “not so Deep Dive” (i.e. a brief overview)

# JSL Application Development

## Concluding Comments

The primary purpose of this exercise was to illustrate the power of XPath. Hopefully, that objective has been achieved.

Note that XPath v1.0 is the currently supported version. Also, JMP provides an `XPath Query(…)` function that is intended for querying XML strings of any origin. `XPath Query(…)` takes a single argument which is the XML string to be queried.

# The Rest: #3 - #5

## Namespaces, Matrices, & Associative Arrays/Lists

# JSL Demo

## Associative Arrays & Matrices

# JSL Application Development

## References


1. Clark, J., & DeRose, S., XML path language (XPath) v1.0, W3C Recommendation (1999), See http://www.w3.org/TR/xpath.html.
2. Morgan, J., “Expression Handling Functions: Unravelling the Expr(), NameExpr(), Eval(), … Conundrum (2010),” JMPer Cable, Issue 26, 15-19.
3. Pressman, R., “Software Engineering: A Practitioners Approach (2009),” McGraw-Hill.
4. SAS Institute, Inc., JMP Scripting Guide, Cary, NC: SAS Institute, Inc.
5. Sebesta, R., “Concepts of Programming Languages (1999),” Addison Wesley.

# Thank You