

Microsoft Power BI

Intro to M Programming


Ted Pattison

Principal Program Manager

Power BI Customer Advisory Team (PBICAT)


Welcome to Power BI Dev Camp

- Power BI Dev Camp Portal - <https://powerbidevcamp.net>




Home | Camp Sessions ▾ | Videos | Camper Resources | About |


Home > Camp Sessions > Session 08: Intro to M Programming

 **Session 08: Intro to M Programming**


This camp session will provide a fast-paced primer for writing Power BI query logic using the M programming language. The goal of this session is to give campers a stronger foundation for working directly with M code in the Advanced Editor when design queries for datasets in Power BI Desktop as well as for dataflows in the browser. Campers will learn how to design efficient queries using query folding as well as how to clean text values using the Text.Select function. Campers will be introduced to writing queries using complex M datatypes including lists, records, tables, functions and user-defined types and optimizing query logic using query folding. The session will demonstrate real-world examples of using dataset parameters, performing calculations across rows during query execution and writing reusable function queries. The session will also discuss how to avoid using dynamic datasources and privacy levels that prevent your queries from being able to run in the Power BI Service.


 **Session Prerequisites**

Campers should have previous experience working with Power Query in Power BI Desktop. It is recommended (but not required) that campers have experience working directly with M code in the Advanced Editor.


 **Session Info**

Date	March 25, 2021
Time	2:00 PM Eastern - 11:00 AM Pacific
Attendee Link	https://aka.ms/PowerBIWebinar03252021


 **Session Links and Resources**

 **PowerPoint Slides for Intro to M Programming**

This is the PowerPoint slide deck used in the presentation on March 25, 2021.

 **Sample PBIX - Intro to M Programming.pbix**

A sample PBIX file with M code samples discussed in this session.

 **Sample PBIX - Query Design Demo.pbix**

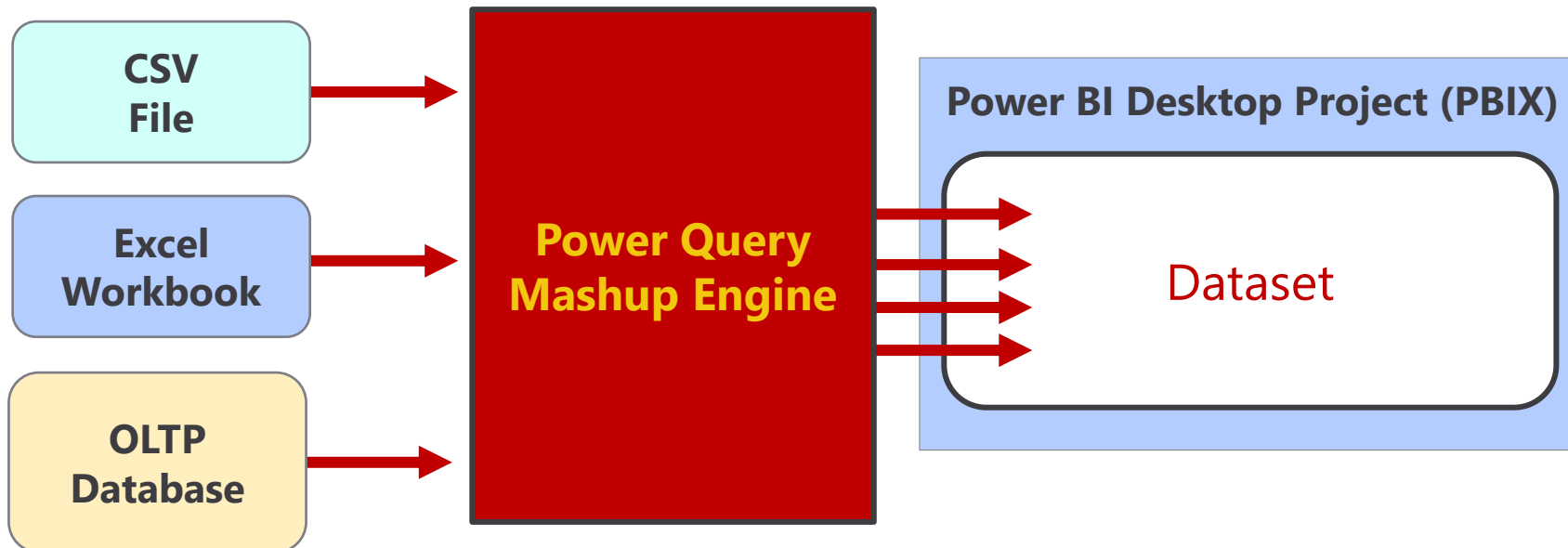
A sample PBIX project which shows how to write and structure queries in an advanced scenario.

Agenda

- The Power Query Mashup Engine
 - M Programming Fundamentals
 - Programming Lists, Records and Tables
 - Understanding Query Folding
 - Choosing Between OData.Feed & Web.Contents
 - Writing Reusable Function Queries
 - Designing with Query Parameters

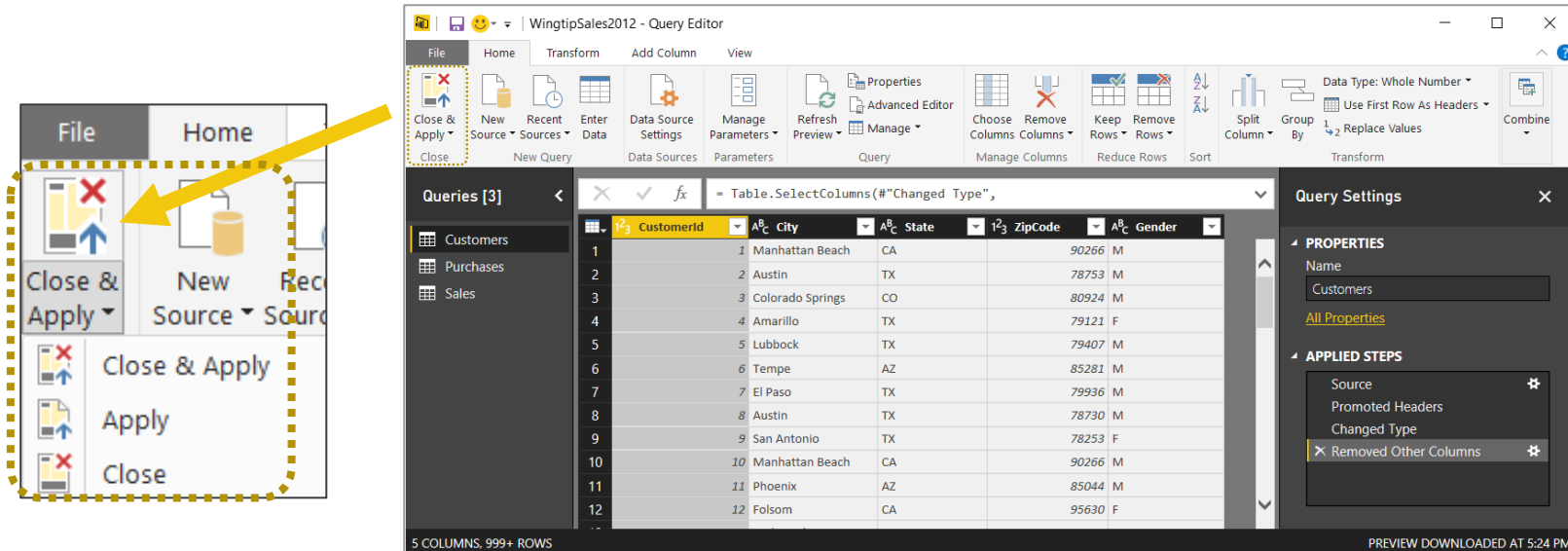
Power Query is an ETL Tool

- ETL process is essential part of any BI Project
 - Extract the data from wherever it lives
 - Transform the shape of the data for better analysis
 - Load the data into dataset for analysis and reporting



Query Editor Window

- Power BI Desktop provides separate Query Editor window
 - Provides easy-to-use UI experience for designing queries
 - Queries created by creating Applied Steps
 - Preview of table generated by query output shown in the middle
 - Query can be executed using Apply or Close & Apply command



Query Steps

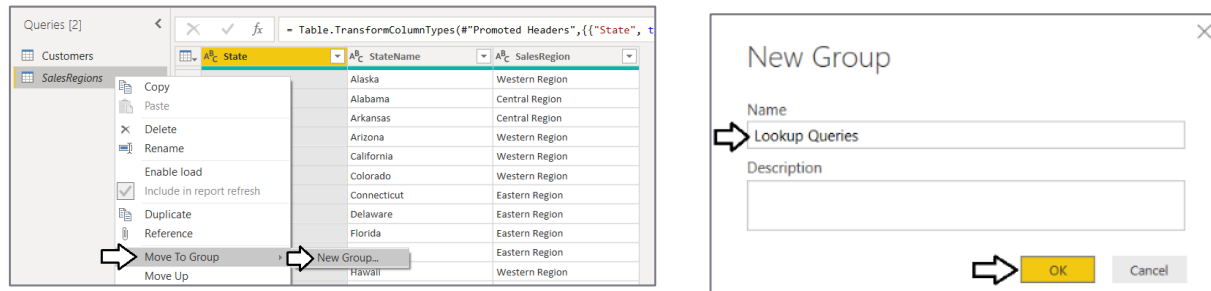
- A query is created as a sequence of steps
 - Each step is a parameterized operation in data processing pipeline
 - Query starts with Source step to extract data from a data source
 - Additional steps added to perform transform operations on data
 - Each step is recorded using M (aka Power Query Formula Language)

The screenshot displays the Microsoft Power Query Editor interface. The top ribbon includes tabs for File, Home, Transform, Add Column, and View. The 'Transform' tab is active, showing options like 'Formula Bar', 'Monospaced', 'Always allow', and 'Show whitespace'. A red dashed box highlights the 'step formula bar' area, which contains the formula: `= Table.ReplaceValue("#Replaced Female Values","M","Male",Replacer.ReplaceText,`. Below the formula bar is a data table with columns: CustomerId, Customer, State, City, Zipcode, and Gender. The table contains 14 rows of customer data. On the right side, the 'Query Settings' pane is open, showing the 'APPLIED STEPS' section. A red dashed box highlights the 'sequential list of steps for query', which includes: Source, Navigation, Removed Other Columns, Merged Columns, Reordered Columns, Replaced Female Values, Replaced Male Values (selected), Changed Type, and Added Conditional Column. A yellow callout box points to the 'step formula bar' with the text 'step formula bar'. Another yellow callout box points to the 'APPLIED STEPS' list with the text 'sequential list of steps for query'.

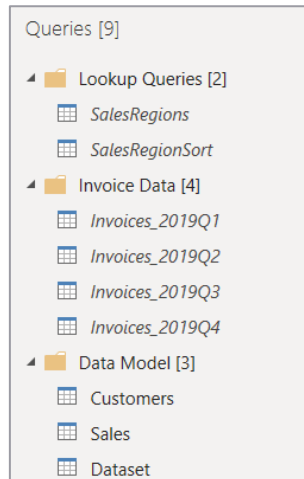
CustomerId	Customer	State	City	Zipcode	Gender
1	Nina Diaz	CA	Eureka	95501	Female
2	Melinda Carter	CA	Napa	94558	Female
3	Pam Miller	CA	Napa	94558	Female
4	Merle Blackwell	CA	Sacramento	95823	Female
5	Ariel Hale	CA	Sacramento	95818	Male
6	Randy Carter	CA	Sacramento	95818	Male
7	Lillie Hinton	CA	Eureka	95501	Female
8	Ladonna Moody	CA	Napa	94559	Female
9	Buddy McKay	OR	Bend	97701	Male
10	Warren Sykes	CA	Sacramento	95818	Male
11	Jan Rutledge	OR	Portland	97216	Female
12	Dallas Lester	OR	Eugene	97402	Male
13	Matthew Zimmerman	OR	Portland	97220	Male
14	Sheryl Hernandez	CA	Sacramento	95823	Female

Structuring Queries into Folder Groups

- Queries can be organized into folder groups
 - Folder groups can be created for similar types of queries

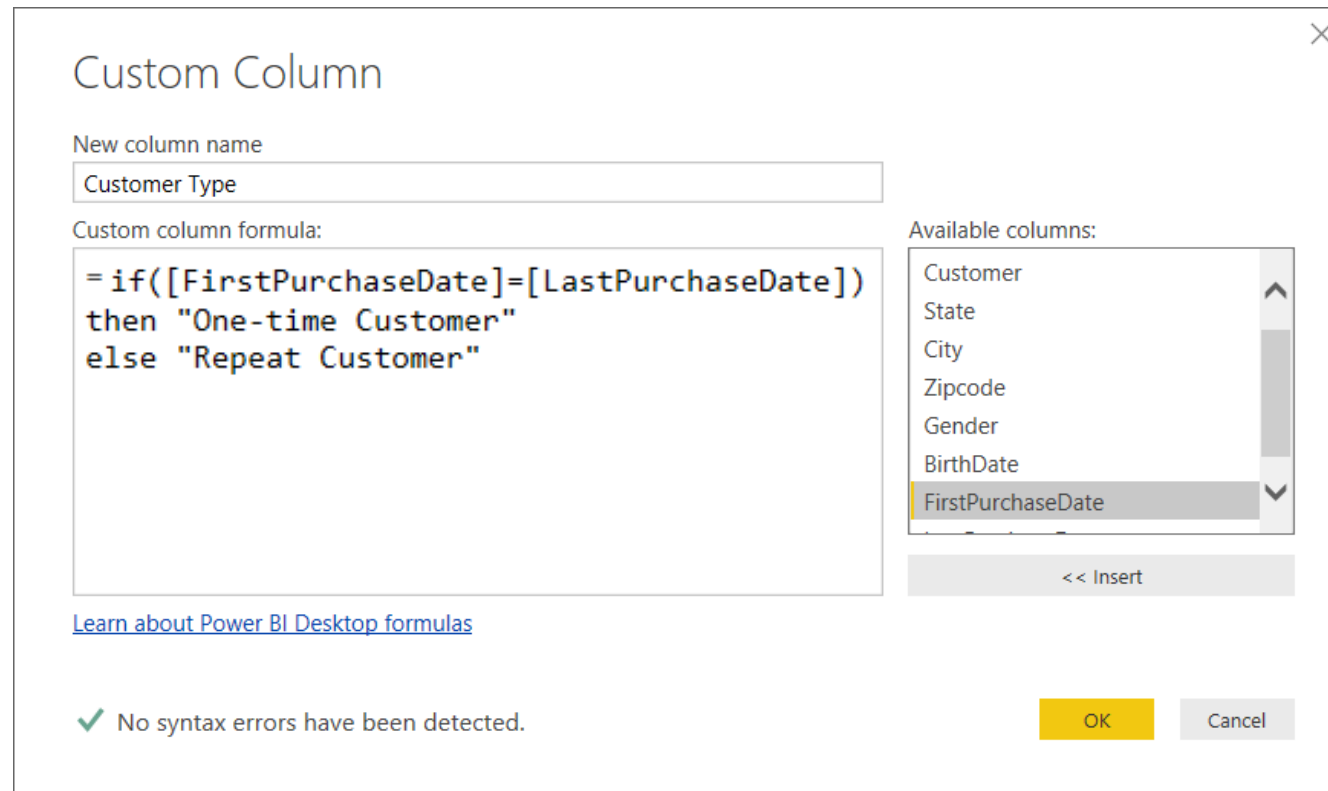


- Makes it easier to manage project with large number of queries



Custom Column Dialog

- You can write M code directly for custom column
 - The Custom Column dialog provides a simple M code editor



The screenshot shows the 'Custom Column' dialog box in Power BI Desktop. It has a title bar with a close button (X). Inside, there's a section for 'New column name' with a text box containing 'Customer Type'. Below that is a 'Custom column formula:' section with a text area containing the M code: `= if([FirstPurchaseDate]=[LastPurchaseDate]) then "One-time Customer" else "Repeat Customer"`. To the right of the formula editor is a list of 'Available columns' including Customer, State, City, Zipcode, Gender, BirthDate, and FirstPurchaseDate (which is highlighted). Below the list is a '<< Insert' button. At the bottom left, there's a green checkmark icon and the text 'No syntax errors have been detected.' with a link to 'Learn about Power BI Desktop formulas'. At the bottom right are 'OK' and 'Cancel' buttons.

Custom Column

New column name
Customer Type

Custom column formula:
`= if([FirstPurchaseDate]=[LastPurchaseDate])
then "One-time Customer"
else "Repeat Customer"`

Available columns:
Customer
State
City
Zipcode
Gender
BirthDate
FirstPurchaseDate

<< Insert

[Learn about Power BI Desktop formulas](#)

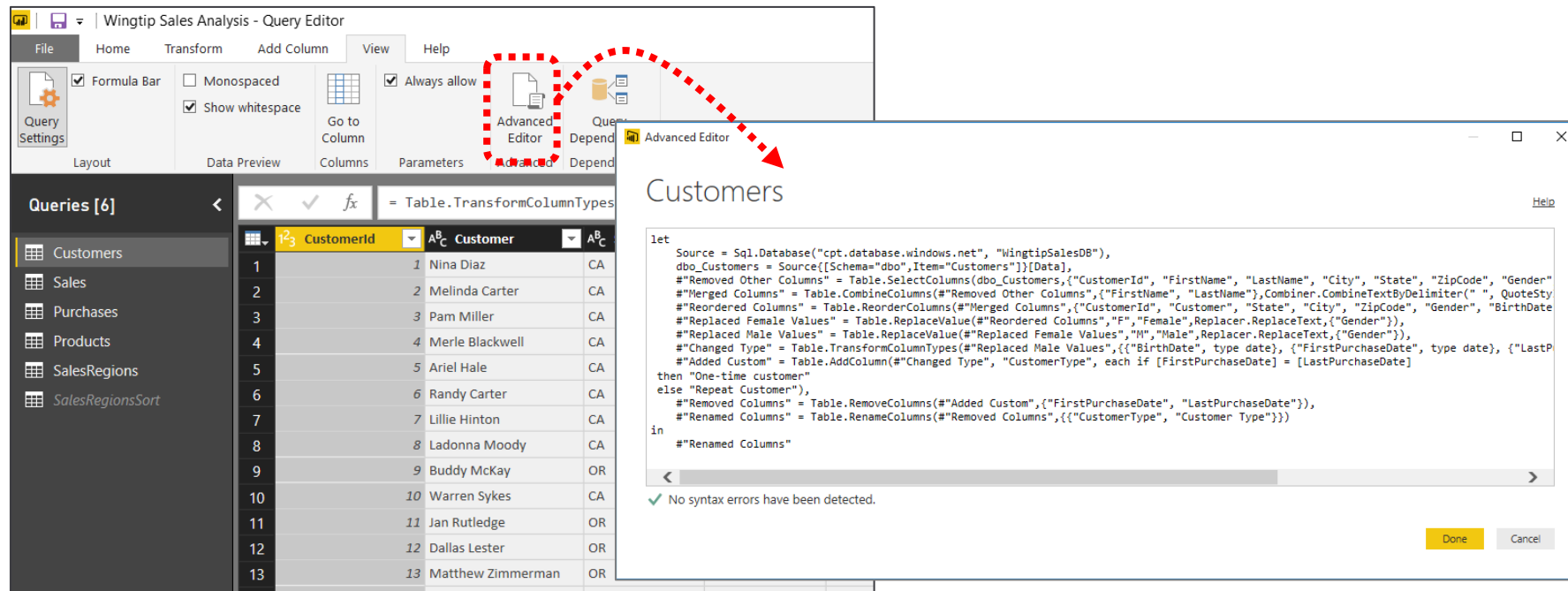
✓ No syntax errors have been detected.

OK Cancel

Advanced Editor

or more correctly - The Simple Editor for Advanced Users

- Power BI Desktop based on "M" functional language
 - Query in Power BI Desktop saved as set of M statements in code
 - Query Editor generates code in M behind the scenes
 - Advanced users can view & modify query code in Advanced Editor



Why Should You Learn M Programming?

- **Accomplish things that cannot be done in query editor**
 - Working with query functions
 - Performing calculations across rows
 - Navigate to SharePoint list by list title instead of GUID with the ID
- **Author queries and check them into source control system**
 - Add query logic in .m files and store them in GitHub, TFS, etc.
 - Ensure query logic is the same across PBIX projects
- **Stay Ahead of the Pack and Win Admiration of Your Peers**
 - People will think you are buddies with Chris Webb!

Why Should You Learn M Programming?

- Take of the biggest of ELT challenges!



#1 Resource for Anyone Serious About Power Query

<https://blog.crossjoin.co.uk>

 Search

CHRIS WEBB'S BI BLOG

Microsoft Power BI, Analysis Services, DAX, M, MDX, Power Query and Power Pivot

HOME

ABOUT

BOOKS

PRIVACY POLICY

SPEAKING

CONTACT



Parquet File Performance In Power BI/Power Query

MARCH 21, 2021

By Chris Webb

in ADLSGEN2,
PARQUET, POWER
BI, POWER QUERY

1 COMMENT

There has been a lot of excitement around the newly-added support for reading from Parquet files in Power BI. However I have to admit that I was disappointed not to see any big improvements in performance when reading data from Parquet compared to reading data from CSV (for example, see [here](#)) when I first started testing it. So, is Power Query able to take advantage of Parquet's columnar storage when reading data?

FOLLOW BLOG VIA EMAIL

Enter your email address to follow this blog and receive notifications of new posts by email.

Join 15,686 other subscribers

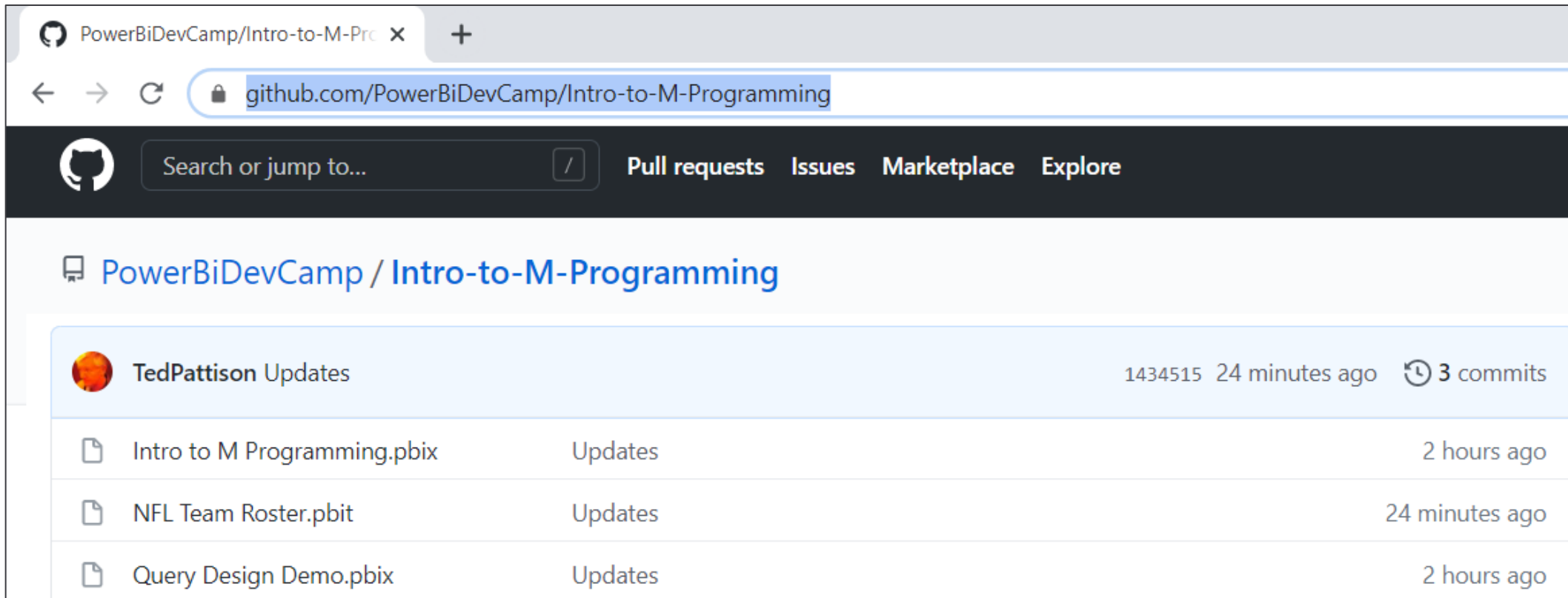
Email Address

Follow

SOCIAL

3 Sample Files Used in This Session

- <https://github.com/PowerBiDevCamp/Intro-to-M-Programming>



The screenshot shows a web browser window displaying the GitHub repository page for `PowerBiDevCamp/Intro-to-M-Programming`. The browser's address bar shows the URL `github.com/PowerBiDevCamp/Intro-to-M-Programming`. The GitHub navigation bar includes a search bar and links for `Pull requests`, `Issues`, `Marketplace`, and `Explore`. The repository name is displayed as `PowerBiDevCamp / Intro-to-M-Programming`. Below the repository name, a commit by `TedPattison` is highlighted, showing `1434515` changes, made `24 minutes ago`, with `3 commits`. A table lists the files updated in this commit:

File	Update Type	Time Ago
<code>Intro to M Programming.pbix</code>	Updates	2 hours ago
<code>NFL Team Roster.pbit</code>	Updates	24 minutes ago
<code>Query Design Demo.pbix</code>	Updates	2 hours ago

Agenda

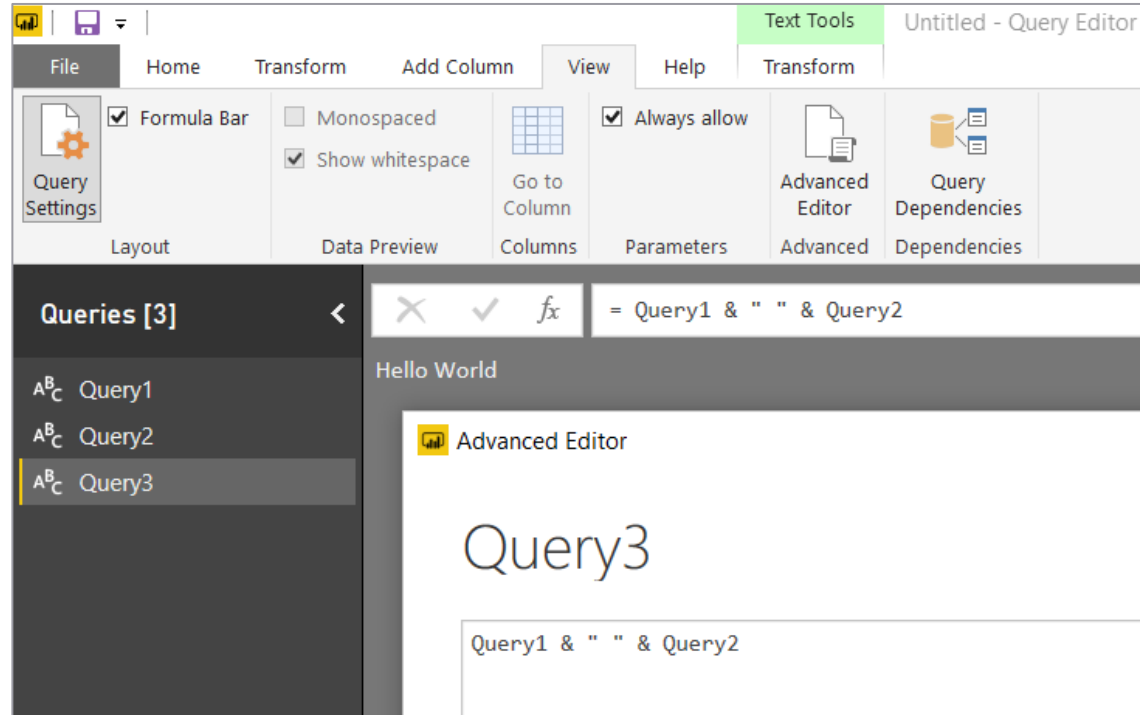
- ✓ The Power Query Mashup Engine
- M Programming Fundamentals
 - Programming Lists, Records and Tables
 - Understanding Query Folding
 - Choosing Between OData.Feed & Web.Contents
 - Writing Reusable Function Queries
 - Designing with Query Parameters

The M Programming Language

- M is a *functional* programming language
 - computation through evaluation of mathematical functions
 - Programming involves writing expressions instead of statements
 - M does not support changing-state or mutable data
 - Every query is a single expression that returns a single value
 - Every query has a return type
- **Get Started with M**
 - Language is case-sensitive
 - It's all about writing expressions
 - Query expressions can reference other queries by name

Referencing Other Queries

- Query can reference other queries by name
 - Every query is defined with a return type



Let Statement

- Queries usually created using **let** statement
 - Allows a single expressions to contain inner expressions
 - Each line in **let** block represents a separate expression
 - Each line in **let** block has variable which is named step
 - Each line in **let** block requires comma at end except for last line
 - Expression inside **in** block is returned as **let** statement value

The screenshot displays the 'Advanced Editor' interface. The main text area shows a query starting with 'let' followed by four lines of code: 'var1 = "Hello",', 'var2 = "World",', 'var3 = var1 & " " & var2,', and 'var4 = Text.Upper(var3)'. These lines are enclosed in yellow boxes, and yellow arrows point from each box to the 'APPLIED STEPS' panel on the right. The 'APPLIED STEPS' panel lists 'var1', 'var2', 'var3', and 'var4' with a small 'X' icon next to 'var4'. The 'PROPERTIES' panel on the right shows the 'Name' property set to 'Hello World' and a link to 'All Properties'. At the bottom, a green checkmark indicates 'No syntax errors have been detected.'

```
let
  var1 = "Hello",
  var2 = "World",
  var3 = var1 & " " & var2,
  var4 = Text.Upper(var3)
in
  var4
```

✓ No syntax errors have been detected.

PROPERTIES
Name
Hello World
[All Properties](#)

APPLIED STEPS
var1
var2
var3
✕ var4

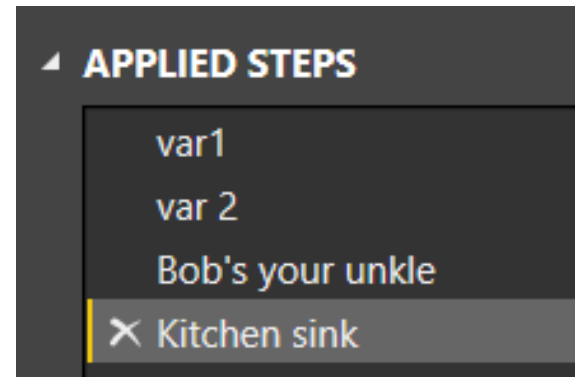
Comments and Variable Names

- M supports using C-style comments
 - Multiline comments created using `/* */`
 - Single line comments created using `//`

```
/*  
  This is my most excellent query  
*/  
let  
  var1 = 42, // the secret of life
```

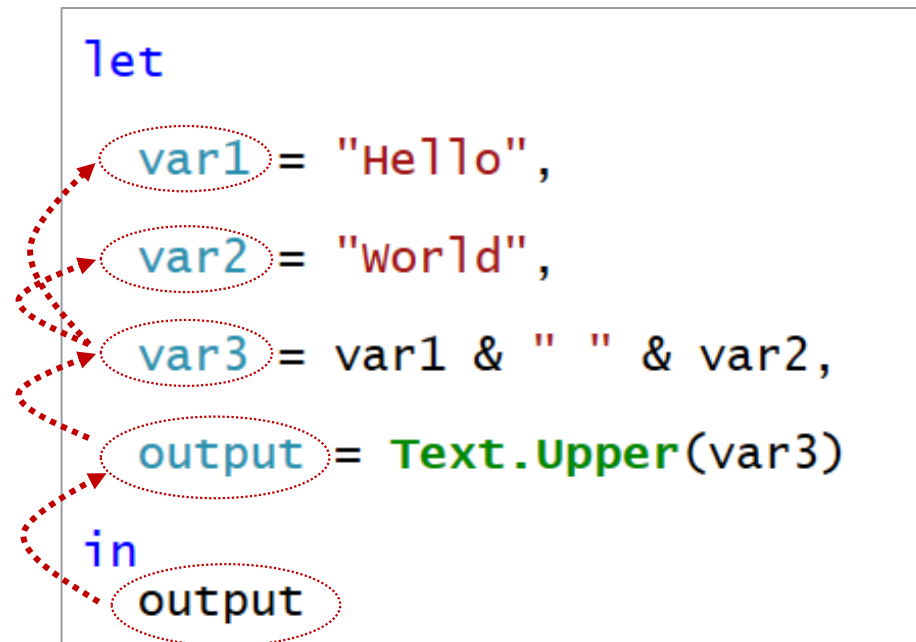
- Variable names with spaces must be enclosed in `#" "`
 - Variable names with spaces created automatically by query designer

```
let  
  var1 = "Spaces in ",  
  #"var 2" = "variable names ",  
  #"Bob's your unkle" = "are evil",  
  #"Kitchen sink" = var1 & #"var 2" & #"Bob's your unkle"  
in  
  #"Kitchen sink"
```



Flow of Statement Evaluation

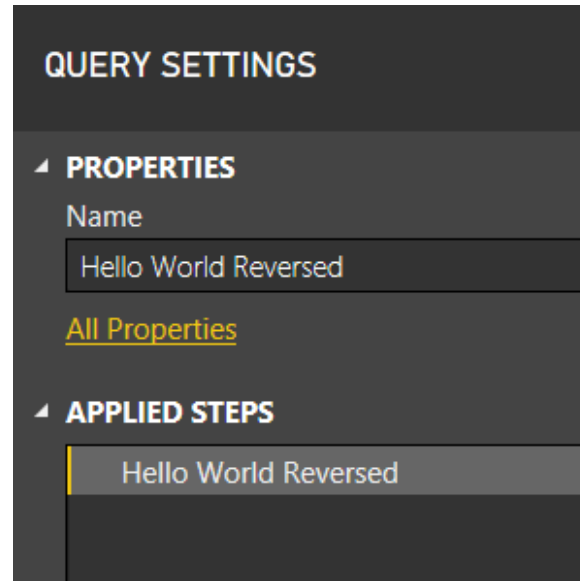
- Evaluation starts with expression inside **in** block
 - Expression evaluation triggers other expression evaluation



Will This M Code Work?

- Yes, the Mashup Engine has no problem with this
 - The order of expressions in **let** block doesn't matter
 - However, the Power Query designer might get confused

```
let
    var4 = Text.Upper(var3),
    var3 = var1 & " " & var2,
    var2 = "world",
    var1 = "Hello"
in
    var4
```



M Type System

- Built-in types

`any, none`

`null, logical, number, text, binary`

`time, date, datetime, datetimezone, duration`

- Complex types

`list, record, table, function`

- User-defined types

- You can create custom types for records and tables

```
CustomerRecordType = type [FirstName = text, LastName = text],
```

Examples of programming with M Datatypes

```
let

// primitives
var1 = 123,      // number
var2 = true,     // boolean
var3 = "hello",  // text
var4 = null,     // null

// creating lists
list1 = {1, 2, 3}, // list of three numbers

// accessing list elements
var5 = list1{1},

// create records
record1 = [ FirstName="Soupy", LastName="Sales", ID=3 ],

// accessing records
var6 = record1[FirstName],

// table
table1 = #table( {"A", "B"}, { {1, 2}, {3, 4} } ),

// creating function
function1 = (x) => x * 2,

// calling function
output = function1(var1)

in
output
```

Initializing Dates and Times

```
// time
var1 = #time(09,15,00),

// date
var2 = #date(2013,02,26),

// date and time
var3 = #datetime(2013,02,26, 09,15,00),

// date and time in specific timezone
var4 = #datetimezone(2013,02,26, 09,15,00, 09,00),

// time durection
var5 = #duration(0,1,30,0),
```


Catching Errors

- Error handling in M done using try .. otherwise

```
try Date.FromText([Raw Date]) otherwise null
```

- Error handling can avoid evaluation errors

```
AddedDateColumn1 = Table.AddColumn(Source, "Date1", each Date.FromText([Raw Date])),  
AddedDateColumn2 = Table.AddColumn(AddedDateColumn1, "Date2", each ( try Date.FromText([Raw Date]) otherwise null ) )
```

- Expression causing errors replace with value such as null

	ABC Raw Date	ABC Date1	ABC Date2
1	Feb 30, 2019	Error	null
2	March 4, 2019	3/4/2019	3/4/2019
3	Cinco de mayo, 2019	Error	null
4	6/4/2019	6/4/2019	6/4/2019
5	07-04-2019	7/4/2019	7/4/2019

Agenda

- ✓ The Power Query Mashup Engine
- ✓ M Programming Fundamentals
- Programming Lists, Records and Tables
 - Understanding Query Folding
 - Choosing Between OData.Feed & Web.Contents
 - Writing Reusable Function Queries
 - Designing with Query Parameters

Lists

- List is a single dimension array
 - Literal list can be created using `{ }` operators
 - List elements accessed using `{ }` operator and zero-based index

```
let
  RatPack = { "Frank", "Dean", "Sammy" } ,
  FirstRat  = RatPack{0} ,
  SecondRat = RatPack{1} ,
  ThirdRat  = RatPack{2} ,
  output = FirstRat & ", " & SecondRat & " and " & ThirdRat
in
output
```

- Use `{ }?` to avoid error when index range is out-of-bounds

```
Rat4 = RatPack{4},    // error - index range out of bounds
Rat5 = RatPack{5}? ,  // no error - Rat5 equals null
```

Text.Select

- Text.Select can be used to clean up text value
 - You create a list of characters to include

```
// take a text value with unwanted characters
input = "!!My text has some @bad things !&^",

// get upper and lower case letters
set1 = {"A".."Z"},
set2 = {"a".."z"},

// get digits 0-9 and convert to text
set3 = List.Transform({0..9}, each Number.ToText(_)),

// add any other allowed characters
set4 = {" ", "-", "_", "."},

// combine all allowed characters in single list
allowedChars = set1 & set2 & set3 & set4,

// call Text.Select to strip out unwanted characters
output = Text.Select(input, allowedChars)
```

Records

- Record contains fields for single instance of entity

```
// create records by using [] and defining fields
Person1 = [FirstName="Chris", LastName="Webb"],
Person2 = [FirstName="Reza", LastName="Rad"],
Person3 = [FirstName="Matt", LastName="Masson"],

// access field inside a record using [] operator
FirstName1 = Person1[FirstName],
LastName2 = Person2[LastName],
```

- You must often create records to call M library functions

```
// create a record to define HTTP request headers
RequestHeaders = [ Accept="application/json",
                  #"OData-MaxVersion"="4.0" ],

// create a second record which contains the first record
OptionsRecord = [ Headers=RequestHeaders ],

// pass the second record as parameter to web.Contents
Response = web.Contents(url, OptionsRecord),
```

Combination Operator (&)

- Used to combine strings, arrays and records

```
// text concatenation: "ABC"  
var1 = "A" & "BC",  
  
// list concatenation: {1, 2, 3}  
var2 = {1} & {2, 3},  
  
// record merge: [ a = 1, b = 2 ]  
var3 = [ a = 1 ] & [ b = 2 ],
```

Table.FromRecords

- Table.FromRecords can be used to create table
 - Table columns are not strongly typed

```
let
    customersTable = Table.FromRecords({
        [ FirstName="Matt", LastName="Masson"],
        [ FirstName="Chris", LastName="Webb"],
        [ FirstName="Reza", LastName="Rad"],
        [ FirstName="Chuck", LastName="Sterling"]
    })
in
    customersTable
```

	ABC 123 FirstName	ABC 123 LastName
1	Matt	Masson
2	Chris	Webb
3	Reza	Rad
4	Chuck	Sterling

ABC
123

Bad, Bad, Bad ☹️

Creating User-defined Types

- M allows you to create user-defined types
 - Here is a user-defined type for a record and a table

```
CustomerRecordType = type [FirstName = text, LastName = text],  
CustomerTableType = type table CustomerRecordType,
```

- User-defined table used to create table with strongly typed columns

```
let  
    CustomerRecordType = type [FirstName = text, LastName = text],  
    CustomerTableType = type table CustomerRecordType,  
    CustomersTable =  
        #table(CustomerTableType, {  
            { "Matt", "Masson" },  
            { "Chris", "Webb" },  
            { "Reza", "Rad" },  
            { "Chuck", "Sterilicious" }  
        })  
in  
    CustomersTable
```

	AB C FirstName	AB C LastName
1	Matt	Masson
2	Chris	Webb
3	Reza	Rad
4	Chuck	Sterilicious

Performing Calculations Across Rows

- Requires adding an index column

	Quarter	\$ Sales	1.2 Index	\$ Running Total
1	2016-Q1	124	0	124
2	2016-Q2	154	1	278
3	2016-Q3	167	2	445
4	2016-Q4	188	3	633
5	2017-Q1	150	4	783
6	2017-Q2	193	5	976
7	2017-Q3	208	6	1184
8	2017-Q4	234	7	1418

PROPERTIES

Name

Sales Running Total

[All Properties](#)

APPLIED STEPS

Source

AddedIndex

✕ AddedCustom

Custom Column

New column name

Running Total

Custom column formula:

= List.Sum(List.Range(AddedIndex[Sales], 0, [Index]+1))

Using Each with Unary Functions

- Many library functions take function as parameters
 - Function passed are often unary functions (*e.g. they accept 1 parameter*)

```
FilteredRows = Table.SelectRows(CustomersTable, (row) => row[CustomerId]<=10 ),
```

- M provides **each** syntax to make code easier to read/write
 - Unary parameter passed implicitly using **_** variable

```
FilteredRows = Table.SelectRows(CustomersTable, each _[CustomerId]<=10 ),
```

- You can omit **_** variable when accessing fields inside record

```
FilteredRows = Table.SelectRows(CustomersTable, each [CustomerId]<=10 ),
```

```
AddedColumn = Table.AddColumn(FilteredRows, "Display Name", each [FirstName] & " " & [LastName])
```

- You must use **_** variable when using **each** with a list

```
MyList = { "Item 1", "Item 2", "Item 3" },  
MyUpperCaseList = List.Transform(MyList, each Text.Upper(_) )
```

List.Generate

- **List.Generate** accepts 3 function parameters

```
MyList = List.Generate( ()=>1, (item)=>(item<=10), (item)=>(item+1) )
```

- You can use **each** syntax for 2nd and 3rd parameter

```
MyList = List.Generate( ()=>1, each _<=10, each _+1 )
```

- You can optionally split functions out into separate expressions

```
let
    StartFunction = ()=>1,
    TestFunction = each _ <= 10,
    IncrementFunction = each _ + 1,
    MyList = List.Generate( StartFunction, TestFunction, IncrementFunction)
in
    MyList
```

	List
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

Agenda

- ✓ The Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ Programming Lists, Records and Tables
- Understanding Query Folding
 - Choosing Between OData.Feed & Web.Contents
 - Writing Reusable Function Queries
 - Designing with Query Parameters

Query Folding

- **Mashup engine pushes work back to datasource when possible**
 - Column selection and row filtering
 - Joins, Group By, Aggregate Operations
- **Datasource that support folding**
 - Relational database
 - Tabular and multidimensional databases
 - OData Web services
- **What happens when datasource doesn't support query folding?**
 - All work is done locally by the mashup engine
- **Things that affect whether query folding occurs**
 - The way you structure your M code
 - Privacy level of datasources
 - Native query execution

Query Folding Example

- When you execute this query in Power BI Desktop...

```
let
    Source = Sql.Database("ODYSSEUS", "WingtipSalesDB"),
    CustomersTable = Source[Item="Customers"][Data],

    // select rows
    FilteredRows = Table.SelectRows(CustomersTable, each ([State] = "FL")),

    // select columns
    ColumnsToKeep = {"CustomerId", "FirstName", "LastName"} ,
    RemovedOtherColumns = Table.SelectColumns(FilteredReaders, ColumnsToKeep),

    // rename columns
    ColumnRenamingMap = { {"FirstName", "First Name"}, {"LastName", "Last Name"} },
    RenamedColumns = Table.RenameColumns(RemovedOtherColumns, ColumnRenamingMap)

in
    RenamedColumns
```

- Mashup Engine executes the following SQL query

```
execute sp_executesql
N'select [_].[CustomerId] as [CustomerId],
        [_].[FirstName] as [First Name],
        [_].[LastName] as [Last Name]
from [dbo].[Customers] as [_]
where [_].[State] = 'FL' and [_].[State] is not null'
```

Native Queries

- No query folding occurs after native query

```
let
    DatabaseServer = "cpt.database.windows.net",
    DatabaseName = "wingtipSalesDB",
    SQL = "SELECT CustomerId, FirstName, LastName" &
        " FROM Customers" &
        " WHERE CustomerId <= 10" &
        " ORDER BY LastName, FirstName" ,
    Source = Sql.Database( DatabaseServer, DatabaseName , [Query=SQL] ),
    output = Source
in
    output
```

CHRIS WEBB'S BI BLOG

Microsoft Power BI, Analysis Services, DAX, M, MDX, Power Query and Power Pivot

[HOME](#) [ABOUT](#) [BOOKS](#) [PRIVACY POLICY](#) [SPEAKING](#) [CONTACT](#)



Query Folding On SQL Queries In Power Query Using Value.NativeQuery() and EnableFolding=true

FEBRUARY 21, 2021

By Chris Webb
in M, POWER BI,
POWER QUERY

FOLLOW BLOG VIA EMAIL

Enter your email address to follow this blog and receive notifications of new posts by email.

Join 15,686 other subscribers

Email Address

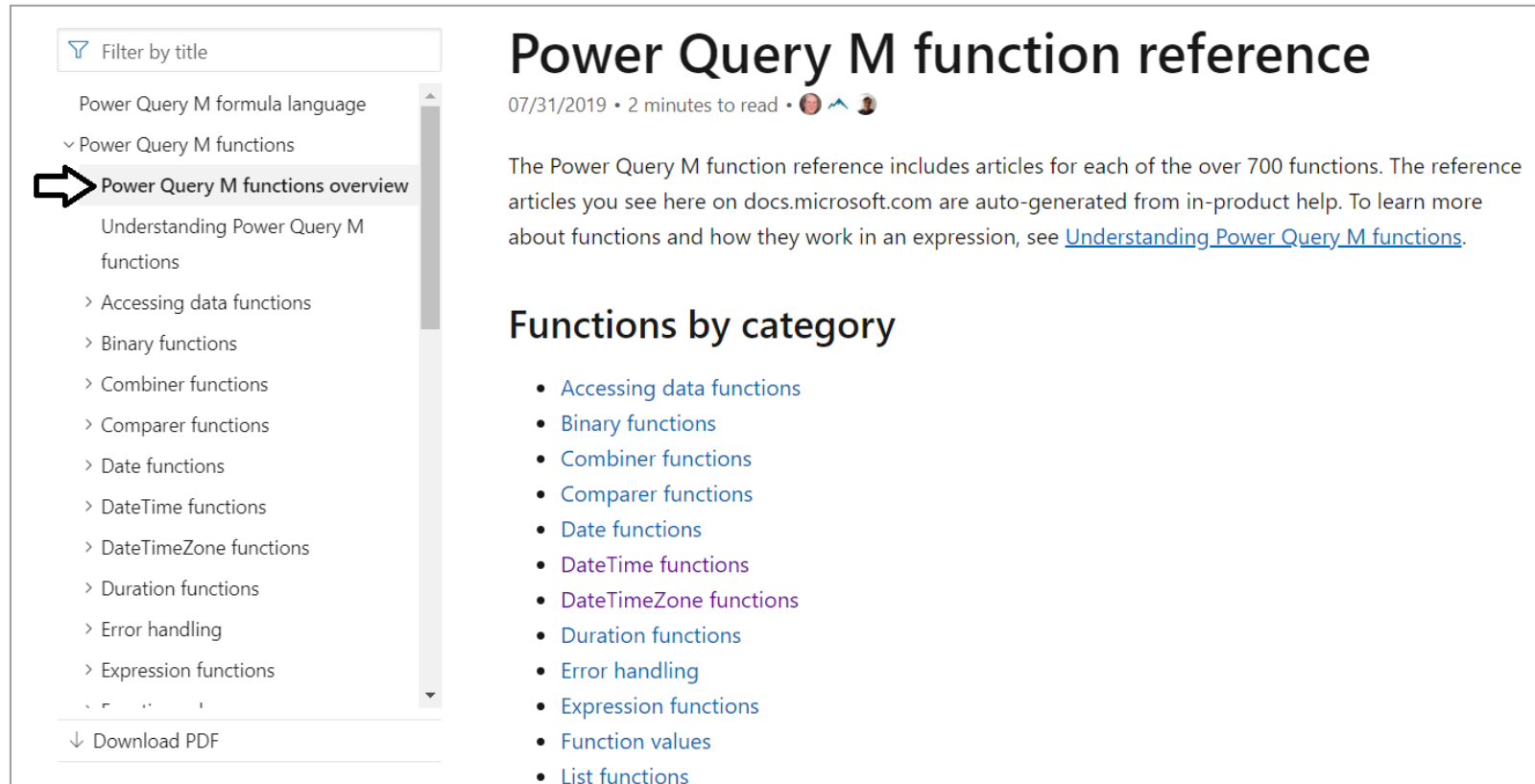
```
1 let
2     Source = Sql.Databases("localhost"),
3     AdventureWorksDW2017 = Source
4         {[Name = "AdventureWorksDW2017"]}
5         [Data],
6     RunSQL = Value.NativeQuery(
7         AdventureWorksDW2017,
8         "SELECT EnglishDayNameOfWeek FROM DimDate",
9         null,
10        [EnableFolding = true]
11    ),
12    #"Filtered Rows" = Table.SelectRows(
13        RunSQL,
14        each (
15            [EnglishDayNameOfWeek] = "Friday"
16        )
17    )
18 in
19    #"Filtered Rows"
```


Agenda

- ✓ The Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ Programming Lists, Records and Tables
- ✓ Understanding Query Folding
- Choosing Between OData.Feed & Web.Contents
 - Writing Reusable Function Queries
 - Designing with Query Parameters

M Function Library

- Check out the Power Query M function reference
 - <https://docs.microsoft.com/en-us/powerquery-m/power-query-m-function-reference>



Filter by title

Power Query M formula language

Power Query M functions

Power Query M functions overview

Understanding Power Query M functions

> Accessing data functions

> Binary functions

> Combiner functions

> Comparer functions

> Date functions

> DateTime functions

> DateTimeZone functions


> Duration functions

> Error handling

> Expression functions

Download PDF

Power Query M function reference

07/31/2019 • 2 minutes to read • 

The Power Query M function reference includes articles for each of the over 700 functions. The reference articles you see here on docs.microsoft.com are auto-generated from in-product help. To learn more about functions and how they work in an expression, see [Understanding Power Query M functions](#).

Functions by category

- [Accessing data functions](#)
- [Binary functions](#)
- [Combiner functions](#)
- [Comparer functions](#)
- [Date functions](#)
- [DateTime functions](#)
- [DateTimeZone functions](#)
- [Duration functions](#)
- [Error handling](#)
- [Expression functions](#)
- [Function values](#)
- [List functions](#)

Accessing Data using OData.Feed

- OData.Feed can pull data from OData web service
 - OData connector assists with navigation through entities
 - OData connector support query folding

```
let
    Source = OData.Feed("http://subliminalsystems.com/api/"),
    // get Customers table
    CustomersTable = Source{[Name="Customers",Signature="table"]}[Data],
    // select columns
    ColumnsToKeep = {"CustomerId", "FirstName", "LastName", "City", "State", "Zipcode", "Gender", "BirthDate"},
    RemovedOtherColumns = Table.SelectColumns(CustomersTable, ColumnsToKeep),
    // select rows
    FilteredRows = Table.SelectRows(RemovedOtherColumns, each [CustomerId] <= 10),
    // perform other transforms
    ReplacedValue = Table.ReplaceValue(FilteredReaders, "F", "Female", Replacer.ReplaceText, {"Gender"}),
    ReplacedValue1 = Table.ReplaceValue(ReplacedValue, "M", "Male", Replacer.ReplaceText, {"Gender"}),
    ChangedType = Table.TransformColumnTypes(ReplacedValue1, {"BirthDate", type date}),
    MergedColumns = Table.CombineColumns(ChangedType, {"FirstName", "LastName",
        Combiner.CombineTextByDelimiter(" ", QuoteStyle.None),
        "Customer"})
in
    MergedColumns
```

- OData makes extra calls to acquire metadata
 - Let's look at the execution of this query using Fiddler

Web.Contents

- Can be more efficient than OData.Feed
 - You can pass OData query string parameters (e.g. \$select)

```
let
    // create REST URI for OData source
    Source = "http://subliminalsystems.com/api/Customers?" &
        "?$select=CustomerId,FirstName,LastName,City,State,Zipcode,Gender,BirthDate" &
        "&filter=(CustomerId+1e+10)",

    // create options record for calling Web.Contents
    OptionsRecord = [Headers=[Accept="application/json;odata=nometadata",
        #"OData-MaxVersion"="4.0"]],

    // call Web.Content to make call across network
    WebContents = Web.Contents(Source, OptionsRecord),

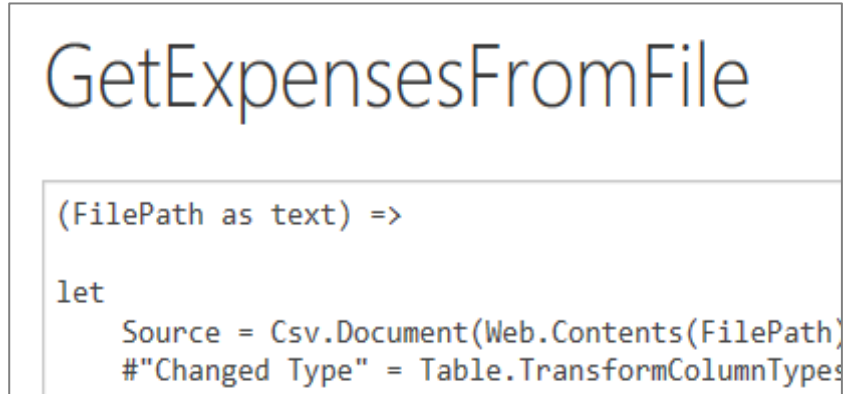
    // deal with JSON dataset return by Web.Contents
    JsonDocument = Json.Document(WebContents),
    RecordList = Record.ToTable(JsonDocument){1}[Value],
    Table = Table.FromList(RecordList, Splitter.SplitByNothing(), null, null, ExtraValues.Error),
    ColumnsToExpand = {"CustomerId", "FirstName", "LastName", "City", "State", "Zipcode", "Gender", "BirthDate"},
    ExpandedColumns = Table.ExpandRecordColumn(Table, "Column1", ColumnsToExpand, ColumnsToExpand),
```

Agenda

- ✓ The Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ Programming Lists, Records and Tables
- ✓ Understanding Query Folding
- ✓ Choosing Between OData.Feed & Web.Contents
- Writing Reusable Function Queries
 - Designing with Query Parameters

Understanding Function Queries

- Query can be converted into reusable function
 - Requires editing query M code in Advanced Editor
 - Function query defined with one or more parameters



The screenshot shows a function query named 'GetExpensesFromFile' defined in the Advanced Editor. The function takes a parameter 'FilePath as text' and returns a table. The code is as follows:

```
GetExpensesFromFile  
  
(FilePath as text) =>  
  
let  
    Source = Csv.Document(Web.Contents(FilePath))  
    #"Changed Type" = Table.TransformColumnTypes(Source, {})
```

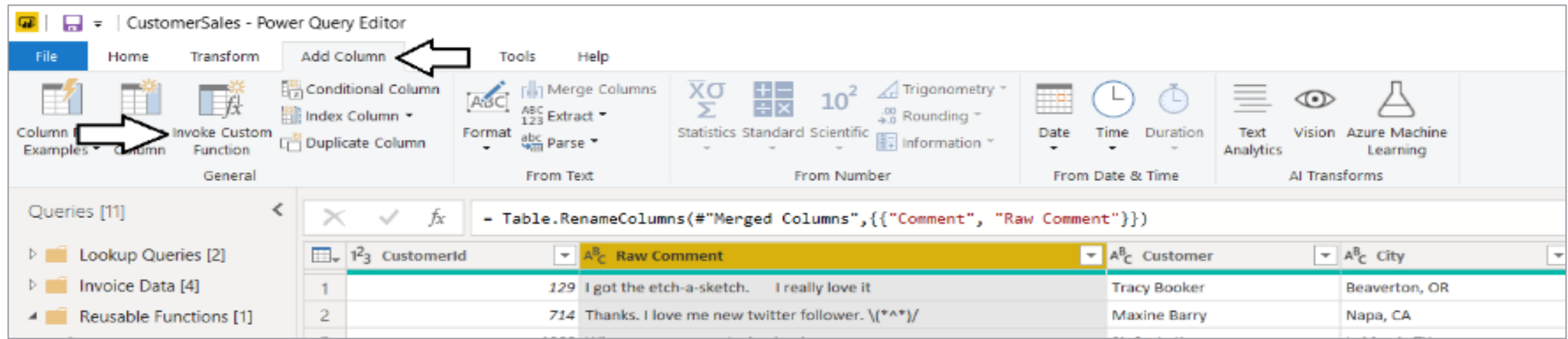
- Function query can be called from other queries
- Function query can be called using Invoke Custom Function
- Function query can't be edited with visual designer

Creating a Function Query

- Requires adding parameter list



Calling a Function Query



The screenshot shows the Power Query Editor interface for a file named 'CustomerSales'. The 'Add Column' ribbon is active, and the 'Invoke Custom Function' button is highlighted with a black arrow. Another black arrow points to the formula bar, which contains the M code: `= Table.RenameColumns(#"Merged Columns",{{"Comment", "Raw Comment"}})`. Below the formula bar, a table of data is visible with columns: CustomerId, Raw Comment, Customer, and City. The 'Raw Comment' column is highlighted in yellow.

	CustomerId	Raw Comment	Customer	City
1	129	I got the etch-a-sketch. I really love it	Tracy Booker	Beaverton, OR
2	714	Thanks. I love me new twitter follower. \[^\^*\]/	Maxine Barry	Napa, CA

×

Invoke Custom Function

Invoke a custom function defined in this file for each row.

New column name

Comment

Function query

CleanText

input

Raw Comment

OK

Cancel

Agenda

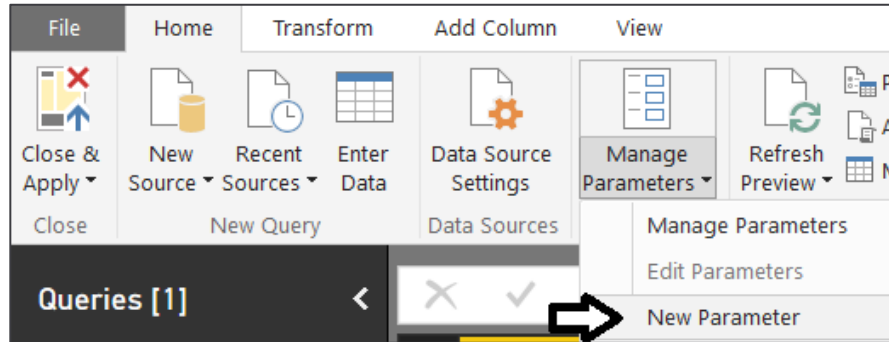
- ✓ The Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ Programming Lists, Records and Tables
- ✓ Understanding Query Folding
- ✓ Choosing Between OData.Feed & Web.Contents
- ✓ Writing Reusable Function Queries
- Designing with Query Parameters

Query Parameters in Power BI Desktop

- **What is a Query Parameter?**
 - Configurable setting for PBIX file
 - Strongly-typed value to which you can apply restrictions
 - Can be referenced from a query
 - Can be referenced from DAX code in data model
 - Query parameters can be updated in Power BI Service
- **Where are Parameters commonly used**
 - To parameterize data source connection details
 - To filter rows when importing data
 - Commonly used together with Template Apps

Creating Query Parameters

- Parameters can be created using Manager Parameters menu



- Parameter properties

- Name
- Description
- Required
- Allowed Values
- Default Value
- Current Value

A screenshot of the 'Parameters' dialog box in Power BI. The dialog shows a list of parameters on the left, with 'Customer State' selected. On the right, the properties for 'Customer State' are displayed:

- Name:** Customer State
- Description:** This parameter is used in the Customers query to filter the customer rows which are loaded into the dataset for the Power BI Desktop project.
- Required:** ☒
- Type:** Text
- Allowed Values:** List of values
- Allowed Values Table:**

1	CA
2	OR
3	WA
4	AZ
5	TX
*	
- Default Value:** CA
- Current Value:** CA

At the bottom right, there are 'OK' and 'Cancel' buttons.

Referencing Parameters in a Query

- Parameters can be referenced inside query
 - Next query execution uses current parameter value

Filter Rows

Basic

Advanced

Show rows where:

And/Or	Column	Operator	Value
	State	equals	<div><div></div>Customer State</div>
And	State		<div><div>A^BC</div></div>

Add Clause

OK

Cancel

Updating Parameters in the Service

The screenshot shows the Power BI Settings interface. The top navigation bar includes a hamburger menu icon, the text "Power BI", and the word "Settings". The left sidebar contains navigation links: Home, Favorites, Recent, Apps, Shared with me, Workspaces, and Wingtip Sales. The main content area has a tabbed interface with "General", "Alerts", "Subscriptions", "Dashboards", "Datasets", "Workbooks", and "Dataflows". The "Datasets" tab is active, showing a list of datasets on the left: "State Sales Report" and "Wingtip Sales Analysis". The "State Sales Report" dataset is selected, and its settings are displayed on the right. The settings include a configuration note, a last refresh timestamp, a refresh history link, and expandable sections for "Gateway connection", "Data source credentials", and "Parameters". The "Parameters" section is expanded, showing three input fields: "DatabaseName" (containing "WingtipSalesDB"), "DatabaseServer" (containing "cpt.database.windows.net"), and "StateFilter" (containing "FL"). A large white arrow points to the "StateFilter" input field. Below the input fields, there are two buttons: "Apply" (highlighted in yellow) and "Discard". A second large white arrow points to the "Apply" button.

Power BI Settings

General Alerts Subscriptions Dashboards **Datasets** Workbooks Dataflows

State Sales Report

Wingtip Sales Analysis

Settings for State Sales Report

This dataset has been configured by TedP@pbdbc2020.onmicrosoft.com.

Last refresh succeeded: Tue Feb 11 2020 10:43:23 GMT-0500 (Eastern Standard Time)
[Refresh history](#)

▶ Gateway connection

▶ Data source credentials

◀ Parameters

DatabaseName

WingtipSalesDB

DatabaseServer

cpt.database.windows.net

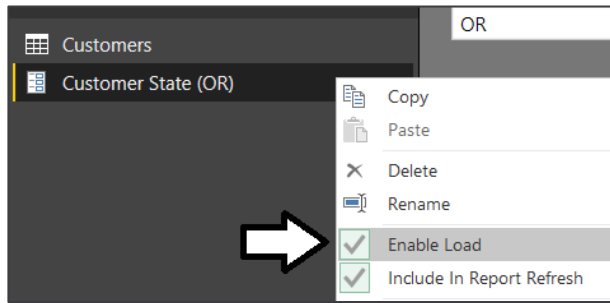
StateFilter

FL

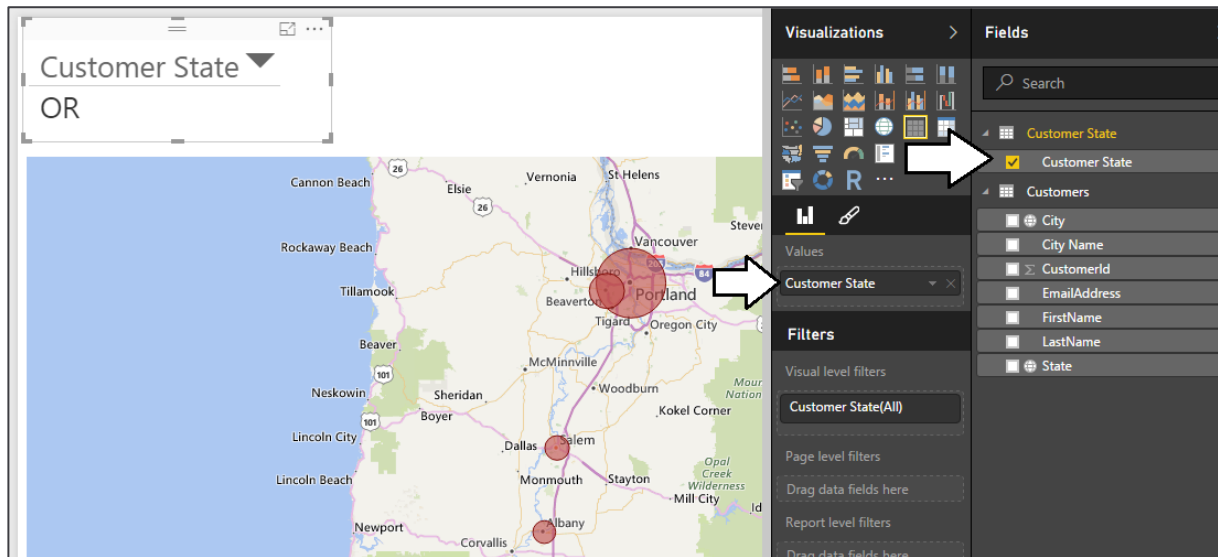
Apply Discard

Making Parameters Available to Data Model

- Configure parameter's Enable Load setting

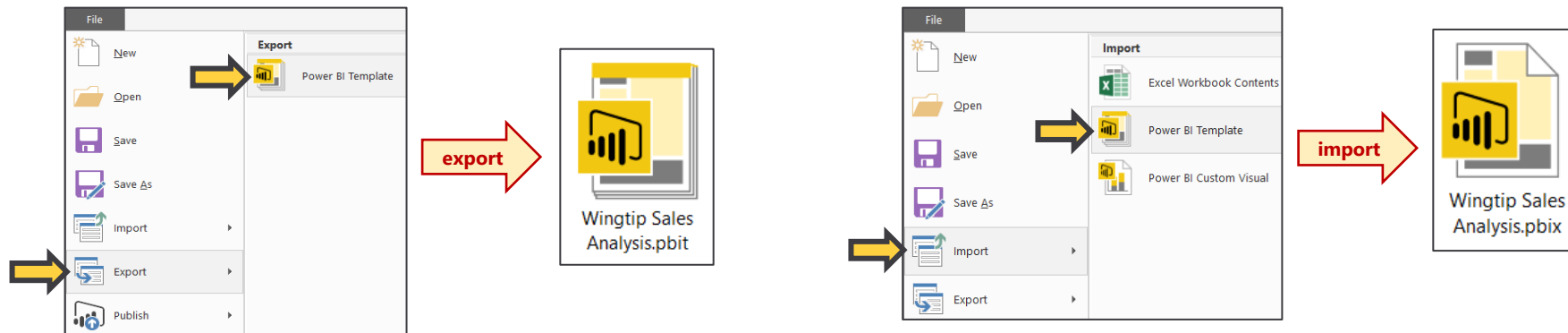


- Parameter becomes visible within fields list in report view



Power BI Project Template Files

- PBIX project can be exported to project template file
 - Template file created with PBIT file extension
 - Generated template files contains everything except for the data
 - PBIT template file can be imported to create new PBIX projects
 - Template files are powerful when used together with parameters
- **How are template files used?**
 - Export PBIX project to create a PBIT template file
 - Import the PBIT template file to create a new PBIX project



Summary

- ✓ The Power Query Mashup Engine
- ✓ M Programming Fundamentals
- ✓ Programming Lists, Records and Tables
- ✓ Understanding Query Folding
- ✓ Choosing Between OData.Feed & Web.Contents
- ✓ Writing Reusable Function Queries
- ✓ Designing with Query Parameters

Questions?

Microsoft Power BI