Table of Contents

Module 1: DataWeave Fundamentals—Review++

Walkthrough 1-1: Import a basic DataWeave based Mule project into Anypoint Studio

Import the starter project

1. Start Anypoint Studio
2. Create a new workspace
3. Import the apdw2-flights-starter.jar project under the studentFiles/mod01

Create a new project

Creating a new project and copying only the files you minimally need for the class helps in containing the "noise" that is introduced with starter project. Additionally there is the extra benefit of not having to deal with students who are having issues with the started project.

4. Create a new project and call it dataweave
5. From the apdw2-flights-starter copy the following files over to the new project:
    a. `src/main/resources/airportInfoTiny.csv` to `src/main/resources`
    b. `src/main/resources/examples/mockdata/deltaSoapResponsesToAllDestinations.xml` to `src/test/resources`
    c. `src/test/resources/flight-example.json` to `src/test/resources`

Walkthrough 1-2: Fundamentals Review++

In this WT the goal is to attempt to bring everyone at the same level by (1) reviewing fundamentals and (2) illustrating features of DW that we will be using throughout the class

Create the flow set the metadata

1. Rename the `dataweave.xml` to `mod1.xml`.
2. Create a new flow named `mod1.review++`
3. Drop a DW (aka Transform Message) to the process area of the flow
4. Define the payload input metadata to the `src/test/resources/flight-example.json`, set the name of the type to `flight_json`
5. Edit the sample data
6. Turn on the preview
7. Change the output to JSON

Construction

8. The semantics of `{}`
    a. What is the meaning of `{}` in DW?
        i. Object creation
9. The semantics of `[]`
    a. What is the meaning of `[]` in DW?
        i. Array creation

Field access

10. Three different ways of accessing the field airline out of the input payload
    a. How can I access the field airline out of the input payload?
        i. `payload.airline`
        ii. `payload['airline']`
        iii. `payload[0]`
11. Objects internally are represented as arrays—field access is a façade

String concatenation

12. Ways to concatenate strings
    a. How can you concatenate strings?
        i. `"The flight is operated by " ++ payload.airline`
        ii. `"The flight is operated by $(payload.airline)"`
13. The `$()` enclosed inside a string acts as an expression evaluator where the result is cast to a `String` and concatenated in place.
    a. `"The flight is operated by $( 1 + 1 )`

Expression chaining

14. Create an array of integers to explain what expression chaining is
    a. Do you know what expression chaining is?

        i. [2,5,3,7,8] map $+1 map $-1
    b. We learned all about expression chains in elementary math!
        i. 1 + 2 – 3
    c. Similarly, to math functional languages like DW use chains to compose programs where the result of one expression is fed into the next and so on.
15. This is a good opportunity to briefly talk about the map function
    a. Do you know what map is?
        i. map is a function
        ii. map uses infix syntax
        iii. map takes two arguments
            1. Left: a collection
            2. Right: a Lambda function (https://en.wikipedia.org/wiki/Lambda_calculus). A lambda function (aka lambda expression, aka anonymous function) is a function that you define (and apply) in place, similar to an anonymous object that you define and construct once.
            3. Returns: another collection where every element from the input collection has been passed as an argument to the Lambda function

Conditional expressions

16. Conditional expressions in DW
    a. `If (true) 1 else 0`
    b. `If (false) 1 else 0`
17. Conditional expressions in DW Objects
    a. `{ (a: 1) if (true), (b: 2) if (false) }`
18. Testing for field membership
    a. payload.airline?
    b. payload.destination?

Array access and Ranges

19. Ranges
    a. How can I get the last element from an array?
        i. [1,5,3][-1]
    b. Do you know what ranges are?
        i. `[2,5,6,3,8,9][1 to 3]`
        ii. `[2,5,6,3,8,9][-3 to -2]`
        iii. `(0 to 100)[-1 to 0]`
        iv. `"ABCDEFG"[-1 to 0]`

Common functions we will be using

20. typeOf, sizeOf, contains, is
    a. typeOf is great when debugging in order to identify the type of data we work with, we will use it a few times for the duration of our class.
        i. typeOf({})
        ii. typeOf([])
    b. sizeOf
        i. sizeOf({})
        ii. sizeOf({a: 1})
        iii. sizeOf(0 to 100)
        iv. sizeOf("ABC")
    c. contains
        i. [2, 6, 4] contains 2
        ii. "ABCD" contains "BC"
    d. Is will be our bread and butter when testing for membership of a value into a type
        i. {} is Object
        ii. [] is Array

Transform XML to JSON

21. Create a new flow and name it mod1.xml2json
22. Drop a DW to the process area
23. Set the input payload metadata to src/test/resources/deltaSoapResponsesToAllDestinations.xml, name the new type to flights_xml
24. Edit the sample data
25. Turn on the preview

26. Change the output to JSON
27. Replace the {} to payload
28. Explore the structure and focus on the objects created with fields of return repeating
    a. Is this a valid JSON data structure?
        i. Nope it is not!
29. Change the output to application/dw

    Application/dw is best used when you want to see how the expression and the input sample data are represented internally by the DW interpreter. It is best used when you are debugging your code.

    a. Is this now a valid DW data structure
        i. Yes it is because DW allows for objects with repeating fields because we need to accommodate XML
    b. How this is possible
        i. it is possible because objects internally are represented as arrays! It's actually ingenious, one transformation language with many build-in supported formats!!
30. Transform the data into a collection
    a. What kind of an expression do you think we need to create an appropriate JSON/DW output?
    b. The goal here is to reform a collection, how can I reform a collection of what DW interprets as objects within objects containing repeating fields?
        i. payload..*return
        ii. payload.findFlightResponse.*return
        iii. ns ns2 http://soap.training.mulesoft.com/

            ---

            payload.ns2#findFlightResponse.*return

            The namespace is important and should not be ignored especially if you foresee changes to this format w.r.t. namespaces. In this particular case we are only interested in the return elements which have no namespaces, as such we can use any of the fore mentioned expressions to reform a collection
31. Go to the first element in the sample data under the return tag
32. Add another return tag with a simple value

```
<return>
    <airlineName>Delta</airlineName>
    <code>A1B2C3</code>
    <departureDate>2018/03/20</departureDate>
    <destination>SFO</destination>
    <emptySeats>40</emptySeats>
    <origin>MUA</origin>
    <planeType>Boing 737</planeType>
    <price>400.0</price>
    <return>10</return>
</return>
```

    a. Show that ..* does a breadth-first recursive search
    b. Show that .* just searches one level down
33. Show the alternate syntax of payload..&return that retrieves the return field in addition to the actual values.
34. Remove the extra return tag you added previously
35. Revert back to payload..*return
36. Go to the preview, select all and copy
37. Create a new file under src/test/resources and call it, flights-example.json

Transform JSON to XML

38. Create a new flow and name it mod1.json2xml
39. Drop a DW to the process area
40. Set the input payload metadata to src/test/resources/flights-example.json, name the new type to flights_json
41. Edit the sample data
42. Turn on the preview
43. Change the output to XML
44. Replace the {} to payload
45. Cannot coarse an array … to a String
    a. What is the meaning of this error?
        i. The problem lies with XML not having any knowledge of arrays but just repeating elements to indicate sequences. No other format that I know of has such semantics, other formats have knowledge of arrays.
        ii. We need to eliminate the arrays
46. Set the output application/dw

47. Eliminate the array by enclosing payload within {(}) i.e. {(payload)}
    *The semantics of () are the usual precedence operators, however **the semantics of parenthesis change when they appear on their own within {} enclosing objects or arrays of objects** to the following: **Break every single object into pairs of keys and values**. The outer {} are there to construct a new object from all the pairs of keys and values. Hence why we end up with single object containing all the keys and their associated values for each object in the collection.*
48. Organize each of the objects inside the array with their own field
    a. How could we organize our data so that even without the collection we can distinguish every single element?
    b. What if before we eliminate the collection, we introduce an object with a single field containing a single object?
        i. {(payload map flight: $)}
49. Set the output to application/xml
50. Trying to output second root
    a. What is the problem?
51. Add a root element
    a. flights: {(payload map flight: $)}
52. Now this elimination of arrays when the output is application/xml is so common that DW 2.0 provides us with a shortcut:
    a. flights: flight: payload

Module 2: Organizing DataWeave Code with Variables and Functions

Walkthrough 2-1: Organize DataWeave code with variables and functions

Create a new flow

1. Create a new Mule Configuration file and name it mod23, it will contain the solutions to all the WTs from modules 2 and 3
2. Create a new flow named `mod23.functions`
3. Drop a DW to the process area of the flow
4. Define the payload input metadata to the `flights_xml`
5. Edit the sample data
6. Turn on the preview
7. Change the output to application/dw
8. Change the body of the expression to payload..*return

Create a variable

9. Create a variable visible throughout the DW expression that contains a static value indicating the total seats of the flights
   var theTotalSeats = 400
10. Introduce the theTotalSeats variable to the expression by adding another field to the result set
    a. Option 1
    ```
    payload..*return map ($ ++ {
          totalSeats: theTotalSeats
    })
    ```
    b. Option 2
    ```
    payload..*return map {
       ($),
       totalSeats: theTotalSeats
    }
    ```

++ is overloaded and can also be used to concatenate objects as you see in the first option. While in the second option we take advantage of the parenthesis semantics when these parenthesis are enclosed inside {}. For the purpose of this document I will use and keep using the second option.

Calculate totalSeats as a function to planeType using `fun`

11. Create and apply a function that calculates the total seats based upon the plane type
    ```
    fun getTotalSeats(pt) = if (
          pt contains "737"
    ) 150 else 300
    ---
    payload..*return map {
          ($),
          totalSeats: getTotalSeats($.planeType)
    }
    ```
    pt is a user defined arbitrary name, denoting the input parameter
12. Change the function expression to allow for the 727 and 707 to be set to 150 seats
    ```
    fun getTotalSeats(pt) = if (
          pt contains "737" or
    ```

```
        pt contains "727" or
        pt contains "707"
) 150 else 300
```

13. Cannot coerce String (737) to Boolean
    a. What is the issue?
        i. Precedence is the issue in this case: or has higher precedence as compared to contains, we need to parenthesize to fix it

    A chunk of the issues you will have when you start creating expressions of reasonable size will be precedence related
14. Discuss issues with the getTotalSeats function
    a. What is the problem(s) with the function we just created?
    b. How can we make it more efficient?
    c. Is there any way I can just compare numbers as opposed to doing a string search?
        i. Get the last three characters from the planeType field and cast it into a number.

    Performance issues in this case are negligible, we just trying hard to find a reason to make another version of the function simple for training purposes ☺

Calculate totalSeats as a function of planeType more efficiently using an anonymous function

15. Create another function named getTotalSeatsL

```
var getTotalSeatsL = (pt) -> pt
```

L is for Lambda, we store an anonymous function to a variable
Return pt to complete the function

16. Apply the function in the expression and get partial results

```
payload..*return map {
        ($),
        totalSeats: getTotalSeatsL($.planeType)
}
```

Applying the function as soon as possible and getting partial results as we code, facilitates unit testing

17. Introduce do {}

```
var getTotalSeatsL = (pt) -> do {
        pt
}
```

At its simplest do does nothing other than specify an expression that is to be returned. However, do can contain any number of localized declarations as we find above the --- inside any DW expression (bar the %dw and the output).

18. Declare a localized variable to obtain the plane number as a actual Number

```
var getTotalSeatsL = (pt) -> do {
        var pn = pt[-3 to -1] as Number
        ---
        pn
}
```

19. Re-introduce the conditional expression

```
var getTotalSeatsL = (pt) -> do {
        var pn = pt[-3 to -1] as Number
        ---
        if (
                pn == 737 or
                pn == 727 or
                pn == 707
        ) 150 else 300
}
```

You can use either one of these two ways to create a simple function; however, if you would like to use advanced features such as function overloading and tail-recursion you MUST stick with fun.

Adjust price for currency

20. Create an object that contains currency exchange rates

```
var xes = {
        USD: 1.0,
        EUR: 0.8,
        CAD: 1.2
}
```

21. Create function to calculate the price adjusted for the currency

```
var adjustedFor = (p,c) -> p * xes[c]
```

22. Apply the function into a new field priceUSD injected into the result set
```
payload..*return map {
        ($),
        totalSeats: getTotalSeatsL($.planeType),
        priceUSD: adjustedFor($.price,"USD")
}
```
23. Apply the function in infix syntax
```
payload..*return map {
        ($),
        totalSeats: getTotalSeatsL($.planeType),
        priceUSD: adjustedFor($.price,"USD"),
        priceEUR: $.price adjustedFor "EUR"
}
```
Functions with exactly two arguments get this infix application supported as well. In fact, infix application is encourage because (1) it is natural in its application, (2) no need to use excessive parenthesis, and (3) allows for expression chains
24.

Walkthrough 2-2: Reuse DataWeave transformations

Store DW code in a file

1. Switch to the XML view of your file
2. Navigate under the mod23.functions flow and illustrate how the code is inline
3. Switch back to the graphical view (aka Message Flow)
4. Go to the properties of the DW processor under the mod23.functions
5. Click the Edit current target button (pencil icon)
6. Click the radio button File and type functions in the text field to the right
7. Click OK
From the point of view of the DW properties UI nothing has changed but with this action you have stored the DW code inside a new file under src/main/resources named functions.dwl

Reuse the DW code from the file

8. Create a new flow named `mod23.reuse`
9. Drop a DW to the process area of the flow
10. Switch to the XML view
11. Locate the DW you just created
12. Remove the CDATA tag
```
<![CDATA[%dw 2.0
output application/java
---
{
}]]>
```
13. Remove the closing </ee:set-payload> tag
14. Introduce the closing / to the opening <ee:set-payload /> tag
15. Add the attribute resource to the <ee:set-payload /> tag
```
<ee:set-payload resource="functions.dwl" />
```
This is the only way you could reuse the full transformation, i.e. by modifying the XML. Had you gone inside the UI and attempting to reuse the file, you would be overwriting it.
16. Switch back to the graphical view
17. Open the properties of the DW processor under the mod23.reuse
18. Turn on the preview
    a. Why is this issue we are seeing?
    b. How can we get rid of it?
        i. The error is there because we do not have any metadata set
19. Set the input payload metadata to flights_xml
20. Edit Sample Data
21. …

Walkthrough 2-3: Create and use DataWeave modules

Create a DW module

1. Create a new folder(s) under src/main/resources

2. In the text field type dw/resources
3. Create a new file under dw.resources and call it MyFirstMod.dwl
4. Type on line 1
   ```
   %dw 2.0
   ```
5. Navigate back to the DW processor under mod23.reuse
6. Copy the xes variable and the adjustedFor function
7. Paste to MyFirstMod.dwl under line 1 and save
   ```
   %dw 2.0
   var xes = {
         USD: 1.0,
         EUR: 0.8,
         CAD: 1.2
   }

   var adjustedFor = (p,c) -> p * xes[c]
   ```

Import the module

8. Go back to the DW processor under mod23.reuse
9. Import the new module below the output directive
   ```
   import * from dw::modules::MyFirstMod
   ```
10. Illustrate that the function defined inline takes precedence
    a. Which function takes precedence?
    b. Is it the one from the module or the one defined inline?
        i. You can quickly show it by modifying the inline definition to return a static value
    c. What if we import two modules with the same function name define?
        i. It is the function from the first import that will be in use—students could easily verify on their own.
11. Change the import directive to just import one of the declarations
    ```
    import adjustedFor from dw::modules::MyFirstMod
    ```
12. Add an alias to adjustedFor
    ```
    import adjustedFor as adj4 from dw::modules::MyFirstMod
    ```

Use the module

13. Modify the expression to also add the the priceCAD field using the adj4 alias
    ```
    payload..*return map {
          ($),
          totalSeats: getTotalSeatsL($.planeType),
          priceUSD: adjustedFor($.price,"USD"),
          priceEUR: $.price adjustedFor "EUR",
          priceCAD: $.price adj4 "CAD"
    }
    ```
14. Change priceEUR to make use of the modules full qualified name
    ```
    payload..*return map {
          ($),
          totalSeats: getTotalSeatsL($.planeType),
          priceUSD: adjustedFor($.price,"USD"),
          priceEUR: $.price dw::modules::MyFirstMod::adjustedFor "EUR",
          priceCAD: $.price adj4 "CAD"
    }
    ```
    In fact there is no need to even do an import as long as you use the modules full qualified name where folders, the module name (minus the extension), and the function/variable names are separated by ::
15. …

Module 3: Writing defensive and robust DataWeave

Walkthrough 3-1: Data matcher via overloaded functions

Create a new flow

1. Create a new flow named `mod23.matcher`
2. Drop a DW to the process area of the flow
3. Turn on the preview
4. Switch to the Source Only view
5. Change the output to application/dw

Match the Null type

6. Create a new function and name it, matcher
```
fun matcher(v: Null) = "Null found"
```
Function parameters can have their types specified. Furthermore, in DW there is a type named Null that its only value is null. Using overloaded function capturing the null value is desirable because you can now capture such invocations and handle them, or at the very least raise an error.
We will see in this module how to raise errors and how to capture them.
7. Test the function by passing null in the body
```
matcher(null)
```
8. Test the function by passing the empty array, []
```
matcher([])
```
The issues indicate that we have a type miss-match

Match other types by overloading the function

9. Overload the function using a parameter of type Array
```
fun matcher(v: Null) = "Null found"
fun matcher(v: Array) = "Array of size $(sizeOf(v))"
```
Overloaded functions must use fun! You cannot overload function using the () -> notation
You must have a function for each type you expect in your data
NEVER create an overloaded function where the type is not specified, the behavior is undefined
10. Overload the function using a parameter of type Object and test
```
fun matcher(v: Null) = "Null found"
fun matcher(v: Array) = "Array of size $(sizeOf(v))"
fun matcher(v: Object) = "Object of size $(sizeOf(v))"
---
matcher({})
```
11. Overload using a String type and test
```
fun matcher(v: Null) = "Null found"
fun matcher(v: Array) = "Array of size $(sizeOf(v))"
fun matcher(v: Object) = "Object of size $(sizeOf(v))"
fun matcher(v: String) = "String of size $(sizeOf(v))"
---
matcher("")
```
12. Overload using a Number and test
```
fun matcher(v: Null) = "Null found"
fun matcher(v: Array) = "Array of size $(sizeOf(v))"
fun matcher(v: Object) = "Object of size $(sizeOf(v))"
fun matcher(v: String) = "String of size $(sizeOf(v))"
fun matcher(v: Number) = "Number $(v as String {format: '#.00'})"
---
matcher(10)
```
13. …

Walkthrough 3-2: Data matcher via the match function

Let's do a similar kind of data matching this time using another function called match. We will see that the match function is a little more flexible as compared to the data you can match

Match expression

1. Stay inside the DW of mod23.matcher
2. Change the body of the expression to
```
[] match {
      else -> $
}
```
Match is comparable to a switch statement or a set of nested if-then-else statements
The else -> $ indicates that If there is no match return the element you are matching.
3. Match arrays
   a. Option 1
```
[] match {
   case is Array -> "Array found"
   else -> $
}
```
   b. Option 2

```
[] match {
    case a if (a is Array) -> "Array found"
    else -> $
}
```

The second option is the verbose choice where you can specify a placeholder, a, to contain the element you are matching. It also allows for any kind of Boolean expression or combination thereof.
For the duration of this class I 'll stick to the second option since it's the more flexible

4. Match objects
```
{} match {
      case a if (a is Array) -> "Array found"
      case o if (o is Object) -> "Object found"
      else -> $
}
```
5. Match strings
```
"" match {
      case a if (a is Array) -> "Array found"
      case o if (o is Object) -> "Object found"
      case s if (s is String) -> "String found"
      else -> $
}
```
6. Match numbers
```
10 match {
      case a if (a is Array) -> "Array found"
      case o if (o is Object) -> "Object found"
      case s if (s is String) -> "String found"
      case n if (n is Number) -> "Number found"
      else -> $
}
```

Match a specific value

7. Now match the number 10, the exact number
```
10 match {
      case a if (a is Array) -> "Array found"
      case o if (o is Object) -> "Object found"
      case s if (s is String) -> "String found"
      case n if (n is Number) -> "Number found"
      case n if (n == 10) -> "10 found!!!!!"
      else -> $
}
```
    a. Why are we not seeing "10 found!!!!!"?
    The problem is the placement of this n == 10 case. Matching happened in order of appearance the first case that matches will execute while all cases that follow will be ignored.
8. Move the n==10 case above the n is Number case
```
10 match {
      case a if (a is Array) -> "Array found"
      case o if (o is Object) -> "Object found"
      case s if (s is String) -> "String found"
      case n if (n == 10) -> "10 found!!!!!"
      case n if (n is Number) -> "Number found"
      else -> $
}
```
9. ..

Walkthrough 3-3: Error handling

Now that we know about match let's explore the error handling from within DW.

Create a new flow

1. Create a new flow named `mod23.errors`
2. Drop a DW to the process area of the flow
3. Turn on the preview
4. Switch to the Source Only view

5. Change the output to application/dw

The error 10 / 0

6. Introduce an error
```
%dw 2.0
output application/dw
---
10 / 0
```
What easier than a division by zero error ☺
7. Open the issues dialog and illustrate the error
What if we can avoid from crashing and burning at this point, what if we can capture the error and do something with it?
There is a module that allows for that kind of behavior!

DW documentation

8. Illustrate where the DW documentation is at (http://docs.mulesoft.com)
Everything related to the Anypoint Platform can be found in here, including DW
9. Navigate to the latest Mule runtime documentation
10. Go to the DataWeave documentation (https://docs.mulesoft.com/mule-runtime/4.2/dataweave)
The single largest topic in the documentation is DW
You can learn the basics by visiting the Quickstart and the Language guide sections
11. Illustrate the DataWeave Examples sections
This is a cookbook, it contains some of the most common issues and their solutions you will be exposed to in DW.
12. Illustrate the Reference documentation
Once you know the basics you will use this documentation as a referenced.
As such lets us learn how to read the reference
Talk a little bit about the plethora of the modules available and that more will be coming in the future
13. Go under dw::Core
All functions under dw::Core are available inside a DW processor, there is no need to import.
14. Show the map function documentation and explain it
The signatures of functions are important! The map function expects to the left an Array containing elements of Type T (T is a generic Type), to the right it expects a function that takes elements of type T as an input and returns elements of type R. The result of map is another Array containing elements of type R. This is done by applying, in order, the function to the right to each element in the array to the left.
All signatures can be read like that.

The dw::Runtime module

15. Navigate to the dw::Runtime module
16. Show the try() function
The try() function takes as an input an anonymous function, with no arguments, and returns the TryResult data structure.
Your expression to be evaluated is the one that you pass as the body to the anonymous function
Whether you get an error or not, the TryResult is what will be returned
17. Examine the `TryResult` structure by navigating to the Runtime Types (https://docs.mulesoft.com/mule-runtime/4.2/dw-runtime-types)
The `TryResult` will always contain a success Boolean value indicating whether there was an error or not. If successful, you will have a result field contain the result of your expression. If there is an error, you will have the `error` set field containing the details.

The dw::Runtime::try() function

18. Go to the DW under the mod23.errors flow
19. Illustrate the structure when in error
```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 0)
```
20. Illustrate the structure with successful
```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 2)
```
21. Use match to return the result when successful otherwise the error message
```
%dw 2.0
output application/dw
---
```

```
dw::Runtime::try(() -> 10 / 0) match {
	case tr if (tr.success) -> tr.result
	else -> $.error.message
}
```

In fact, if you know that this is all you want to do you can extrapolate this code into a function to guard against errors

Extrapolate to a function

22. Create the guard function

```
var guard = (fn) -> dw::Runtime::try(fn) match {
	case tr if (tr.success) -> tr.result
	else -> $.error.message
}
```

23. Illustrate the application of guard() with a correct expression

```
guard(() -> 10 / 2)
```

24. Illustrate the application of guard() with errors

```
guard(() -> 10 / 0)
```

Illustrate the orElse() and orElseTry()

25. TBC

Walkthrough 3-4: Partial results

Let's see how we can get partial results now that we know how to capture errors.
How many times you had your DW code fail because one or very few records were malformed?
We will replicate such a scenario by creating an array containing the string representation of dates and then casting them into a date type. One of our dates will be malformed

1. Stay in the mod23.errors flow
2. Create an array of string dates and assigned it in a variable

```
var dates = [
	"11/01/2019",
	"2020-01-31",
	"01/01/2030"
]
```

It is the second date that is malformed, it will fail our transformation

3. Iterate over dates and attempt to cast into a date

```
dates map (
	$ as Date {format: "MM/dd/yyyy"}
)
```

There is an error thrown telling you that the second-string date cannot be casted.

4. Enclose the casting within the guard() function

```
dates map guard(
	() -> $ as Date {format: "MM/dd/yyyy"}
)
```

Now you get results and the TryResult.error.message where the operation failed.

5. Finally filter for Dates and get your partial results while ignoring the rest

```
dates map guard(
	() -> $ as Date {format: "MM/dd/yyyy"}
) filter ($ is Date)
```

In fact, we can build a function that attempts to parse a string-date into a Date time successively trying using different date formats.

6. …

Modules 4 & 5: Operating on Arrays and Objects

I complete these two modules at the end of Day 1 inside a separate Mule Configuration file I name mod45.xml. I find them tedious for the most part and I will use what time I have left to get them done. It usually takes me no more than 40 to 50 min to complete both modules—I do reduce the time allotted if I am running out of time, I have done both in just 30 min.
I do not talk about object creation because I talk about it at the beginning of the class when I attempt to bring all my students at the same level. I also believe constructing object should be done as early as possible not after most of day one is done.

Walkthrough 4-1: ++, +, --, -, operators on objects and arrays

Examples of arrays and objects and how they behave when used against the ++, --, +, and – operators. I usually set the input payload to flight_json and do my quick tests

Follow the instructions from the manual

Walkthrough 4-2: dw::core function on arrays, zip, unzip, flatten (optional)

Follow the instructions from the manual

I will usually skip this WT since its easy and tedious, I may mention briefly what zip, unzip does by taking my students into the manual.

Walkthrough 5-1: dw::core::Objects module

Follow the instructions from the manual

I take my students inside the manual, read the manual page, illustrate in a flow, all functions

Walkthrough 5-2: dw::core::Arrays module

Follow the instructions from the manual

I take my students inside the manual, illustrate a couple of the functions, then I ask the students if there is a anything else they would like me to illustrate from the list.

Module 6: Flights and airports

Walkthrough 6-1: Change field names

I introduce this WT in order to have a use case/WT to start talking about the `mapObject` function, considering the remove fields use case we used to run in DW 1.0 class is no longer relevant.

This is the Mule Configuration file that we will now start solving a use case and keep coming back to this file to apply what we have learned.

Create a new flow

1. Create a new Mule Configuration file and name it mod6
2. Create a new flow named `mod6.flights`
3. Drop a DW to the process area of the flow
4. Define the payload input metadata to the `flights_json`
5. Edit the sample data
6. Turn on the preview
7. Change the output to `application/dw`
8. Change the expression to payload

Create the map

9. Create a variable, `fs2fs`, that contains a map from source field names to target field names

```
var fs2fs = {
      airlineName: "carrier",
      departureDate: "date",
      origin: "start",
      destination: "finish"
}
```

We need to change the fields for our transformation. I do know we can just do it in-place as part of the lambda expression of a `map` function. Creating the variable will allow us to show how it can be done dynamically, I could easily pass such a map through an HTTP request, read it from a file, etc.

What we need to do is iterate over an object!

The `mapObject` function

10. Illustrate `mapObject` over `payload[0]`
    a. Using the build-in placeholders

```
payload[0] mapObject (
      {($$): $}
)
```

    b. Prefix fields with the string `flight-`

```
payload[0] mapObject (
      {("flights-$($$)"): $}
)
```

    c. Using user-defined arguments, add the index as a suffix to the key

```
payload[0] mapObject ( (v,k,i) ->
      {("flights-$(k)-$(i)"): v}
```

```
)
```

Visit the manual page for `mapObject` and explore for yourself the signature of this function, https://docs.mulesoft.com/mule-runtime/4.2/dw-core-functions-mapobject

Change the field names

11. Iterate through the flights and apply the `fs2fs` map

```
payload map (
	$ mapObject (
		(v,k,i) -> {(fs2fs[k]):v}
	)
)
```

What is the issue we are getting?

The issue is there because the `fs2fs` map is not exhaustive—for fields not in the `fs2fs` we get null, hence the errors

How can we fix it?

12. Apply default to the field name

```
payload map (
	$ mapObject (
		(v,k,i) -> {(fs2fs[k] default k):v}
	)
)
```

13. ..

Walkthrough 6-2: Inject airport details in each flight

Explore the CSV file

1. Open the `src/main/resources/airportInfoTiny.csv` to a text editor

   This file contains a set of airport details. In fact, we have an error in our data, PDX appears twice. As in a real situation we are not dealing with correct data all the time, you will be getting bad data that you and I need to fix.
2. Stay in the `mod6.flights` flow
3. Open the DW properties

Parse the CSV airports info file

4. Read and parse the `airportInfoTiny.csv`

```
var airportInfo = readUrl(
	"classpath://airportInfoTiny.csv","application/csv"
)
```

You can comment your existing expression out and just show the contents of the `airportInfo` variable

5. Briefly comment out the expression in the body
6. Add airportInfo in the body to illustrate the parse content
7. Restore the commented expression

Inject the airport info to each flight

8. Chain another `map` function

```
payload map (
	$ mapObject (
		(v,k,i) -> {(fs2fs[k] default k):v}
	)
) map {

}
```

9. Reintroduce the fields from the previous map
10. Introduce a new field, `airportInfo`, and store the airport details per flight, using the `filter` function.

```
payload map (
	$ mapObject (
		(v,k,i) -> {(fs2fs[k] default k):v}
	)
) map {
	($),
	airportInfo: (airportInfo filter (e,i) -> e.IATA == $.finish)[0]
}
```

Functions as values

11. Create and apply a function that does the filtering

```
var airportInfoFilterByIATA = (d) -> airportInfo filter $.IATA == d
```

```
---
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($),
      airportInfo: airportInfoFilterByIATA($.finish)[0]
}
```

12. Create and apply an almost generic filter function
    a. How can I build a generic function to do all types of filtering?
    b. How many arguments do you think the function needs?

```
var genericFilter = (a,f,v) -> a filter $[f] == v
---
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($),
      airportInfo: genericFilter(airportInfo, "IATA", $.finish)[0]
}
```

Is this truly generic?
It is not because the == is hardcoded, what if I want to filter for lt, or ge?

13. Build a truly generic function
    a. How can we define a truly generic function so that we also pass relational/logical operators?
    b. How can you apply the lambda function from within `genericFilterFn`?

```
var genericFilterFn = (a, fn) -> a filter fn($)
---
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($),
      airportInfo: genericFilterFn(airportInfo, (e) -> e.IATA == $.finish)[0]
}
```

In essence we just created filter! Look at the arguments we used—these are the arguments of filter. The only reason for this exercise is to ensure we understand how to create a function that takes anonymous functions as arguments, then apply them.
Functions can also be used to return other functions as their return values!

Curried functions

14. Create a curried function

Nope, there is no relationship to the spice. The name is taken from one of the founders of modern Computer Science Haskell Brooks Curry; https://en.wikipedia.org/wiki/Haskell_Curry
Here's the details on currying; https://en.wikipedia.org/wiki/Currying

```
var genericFilterFnC = (a) -> (fn) -> a filter fn($)
---
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($),
      airportInfo: genericFilterFnC(airportInfo)((e) -> e.IATA == $.finish)[0]
}
```

Two main reasons for creating curried functions:

a. *consider it a factory function that can be used to create new functions, here's two functions that have been created by applying the `genericFilterFnC` to two different arrays! In fact, you can apply the `airportInfoFilter` instead of the inline application we performed above*

```
var airportInfoFilter = genericFilterFnC(airportInfo)
```

```
var flightsFilter = genericFilterFnC(payload)
```

b. *the most important reason for is the ability to **partially apply a function** because you do not have all the arguments at the same time!*

This is one of the topics that will give you a hard time—if you have never done functional programming if your past, this concept of curried functions will challenge you. I have no illusions that you will understand curried functions in this class nor functional programming at large—what my expectation is that you are being exposed and now you can give yourself the ne necessary time to absorb and understand.

One way you can try something simpler for yourselves is to do a curried add function. Very similar to this discussion in stackoverflow—https://stackoverflow.com/questions/36314/what-is-currying

A more efficient transformation

15. Change the transaformation to make it more efficient
    a. How efficient is our current transformation?
    b. What is its BIG-O complexity?
    c. How can we make it more efficient?
16. Comment out the expression
17. Display the `airportInfo` data structure
18. orderBy the `airportInfo` by `IATA`
    a. PDX contains two records

```
var airportInfo = readUrl(
        "classpath://airportInfoTiny.csv","application/csv"
) groupBy $.IATA
```

19. `distinctBy` the `airportInfo` by `IATA`, before the `groupBy` to eliminate the duplicate PDX record

```
var airportInfo = readUrl(
        "classpath://airportInfoTiny.csv","application/csv"
) distinctBy $.IATA groupBy $.IATA
```

20. Remove the array from the `airportInfo`

```
var airportInfo = readUrl(
        "classpath://airportInfoTiny.csv","application/csv"
)
distinctBy $.IATA
groupBy $.IATA
mapObject {($$): $[0]}
```

21. Restore the commented expression and remove the `airportInfo` from the last line of the expression
22. Locate the airport details record more efficiently

```
payload map (
        $ mapObject (
                (v,k,i) -> {(fs2fs[k] default k):v}
        )
) map {
        ($),
        airportInfo: airportInfo[$.finish]
}
```

Clean up the data

23. Remove the `timeZone`, `type`, and `source` fields from the `airportInfo` object
24. Rename the field `airportInfo` to `finish`
    a. What is the problem now?
25. Remove the original `finish` field

```
payload map (
        $ mapObject (
                (v,k,i) -> {(fs2fs[k] default k):v}
        )
) map {
        ($ - "finish"),
        finish: airportInfo[$.finish] -- ["timeZone", "type", "source"]
}
```

26. …

Walkthrough 6-3: Reorder the flights object fields

We will be re-ordering the fields of an object based upon positional indexes.

We can develop a function from within the `mod6.flights` flow but it is best to do it using a separate flow with a new DW to facilitate unit testing.

Create a new flow

1. Create a new flow named `mod6.reorder`
2. Drop a DW to the process area of the flow
3. Define the payload input metadata to the `flight_json`
4. Edit the sample data
5. Turn on the preview
6. Change the output to `application/dw`
   We will be using two new build in functions, pluck and reduce

Learn plunk

7. Demonstrate `plunk`
   a. Get the keys of the payload object
      ```
      payload pluck $$
      ```
   b. Get the values of the payload object
      ```
      payload pluck $
      ```

Learn reduce

8. Demonstrate `reduce`
   a. Calculate the summation using an array of numbers
      ```
      [3,6,2,1] reduce (e, acc=0) -> acc + e
      ```
      You can also use $$ for the accumulator and $ for the current element, albeit I advise to always name them and initialize the accumulator accordingly
   b. Collapse the array and get an object
      ```
      (payload dw::core::Objects::divideBy 2) reduce (
            (e, acc={}) -> acc ++ e
      )
      ```
   c. Dynamically traverse a data structure by passing the path in a string
      ```
      var ds = {
            flights: {
                  flight: payload
            }
      }
      var path = "flights.flight.planeType"
      ---
      (
            path splitBy /\./
      ) reduce (
            (e, acc=ds) -> acc[e]!
      )
      ```
      The `!` is there to eliminate a false positive that DataSense is throwing. Its semantics is that the field will always be present
9. Clear the `reduce` expression and replace it with `payload`

Re-ordering fields

10. Create the reordering function
    The function that takes two arguments, to the left the object to be reordered, to the right an array of positional indexes for the reordering
    Return the object so that we can get results and unit test every single change we perform
    ```
    var reorder = (o,ris) -> o
    ---
    payload reorder [8,2,5,3,7,6,1,0,4]
    ```
11. Create a localized variable to the function that contains an array of the fields of the object in order, return the localized variable
    ```
    var reorder = (o,ris) -> do {
          var fs = o pluck $$
          ---
          fs
    }
    ---
    payload reorder [8,2,5,3,7,6,1,0,4]
    ```

12. Iterate over the reordered indexes and for each object create an object

```
var reorder = (o,ris) -> do {
      var fs = o pluck $$
      ---
      ris map {}
}
---
payload reorder [8,2,5,3,7,6,1,0,4
```

13. Use the `fs` array to retrieve the field, use a static 1 for the value

```
var reorder = (o,ris) -> do {
      var fs = o pluck $$
      ---
      ris map {(fs[$]): 1}
}
---
payload reorder [8,2,5,3,7,6,1,0,4]
```

14. Add the value

```
var reorder = (o,ris) -> do {
      var fs = o pluck $$
      ---
      ris map {(fs[$]): o[$]}
}
---
payload reorder [8,2,5,3,7,6,1,0,4]
```

15. Collapse the array of objects into a single object

```
var reorder = (o,ris) -> do {
      var fs = o pluck $$
      ---
      ris map {
            (fs[$]): o[$]
      } reduce (e, acc={}) -> acc ++ e
}
---
payload reorder [8,2,5,3,7,6,1,0,4]
```

16. Copy the function

Apply the function to mod6.flights

17. Go back to the `mod6.flights` flow and open the DW properties
18. Paste the function right above the --- i.e. the declarative section of your DW code

```
var reorder = (o,ris) -> do {
      var fs = o pluck $$
      ---
      ris map {
            (fs[$]): o[$]
      } reduce (e, acc={}) -> acc ++ e
}
---
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($ - "finish"),
      finish: airportInfo[$.finish] -- ["timeZone", "type", "source"]
}
```

19. Add a new link to our expression chain and reorder the flights object in reverse using a range, i.e. `(7 to 0)`

```
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($ - "finish"),
      finish: airportInfo[$.finish] -- ["timeZone", "type", "source"]
```

```
} map (
      $ reorder (7 to 0)
)
```

20. .

Module 7: Traverse and transform any data structure

Walkthrough 7-1: Recursion and tail-recursion

Create a new flow

1. Create a new Mule Configuration file and name it mod6
2. Create a new flow named `mod7.recsum`
3. Drop a DW to the process area of the flow
4. Turn on the preview
5. Change the output to `application/dw`

Intro to recursion

6. Create the `recsum` function
   We will be creating a new recursive function to calculate the summation of a number.
   ```
   fun recsum(n: Number) = if (n <= 0) 0 else n + recsum(n - 1)
   ---
   recsum(3)
   ```
7. Try it with `recsum(254)`
8. Try it with `recsum(255)`
   The error is Stack Overflow because we are only allowed to recurse a maximum of 256 times.
   You can change the value by the `com.mulesoft.dw.stacksize` Mule Runtime startup configuration option.
   But there is another way we can do away altogether with such stack-overflow errors, this other way is to develop you function such that the very last operation in your function's body is the recursion! Such a function is called tail-recursive and the DW interpreter will optimize it by changing it into a simple "loop"—this way you avoid being hit by the stacksize limit of 256—for more on tail-recursion please consult the following links, https://en.wikipedia.org/wiki/Tail_call, https://stackoverflow.com/questions/33923/what-is-tail-recursion
   ***Warning**: creating tail recursive functions is not easy and will require experience to find the right algorithm in order to have the recursive call as the only operation you find at the end of the function!*

Tail-recursion

9. Create a tail recursive version of `recsum`
   a. What is the last thing that happens in the body of the current function?
   b. How can we change our function to allow for the recursive call to happen at the very end?
   Tail recursion is ONLY supported using the `fun` syntax and not the `var = () ->`
   ```
   fun tailrecsum(n: Number, r: Number = 0) = if (
         n <= 0
   ) r else tailrecsum(n - 1, r + n)
   ```
10. Try `tailrecsum` with numbers larger than 255, in fact try it with 2550, even with 25500!
    ```
    tailrecsum(25500)
    ```
    a. Which version do you prefer from the developer's perspective, from the human perspective?
    b. Which one is easier to read?
    Tail-recursive functions are less readable, less natural as compared with their counterparts. The reason for this should be obvious, with tail-recursion we are trying to accommodate the interpreter/compiler, i.e. the machinery, and not the human.
    IMHO if you can do without the tail-recursive version of your function then you should do away with it.
11. ..

Walkthrough 7-2: Recursive flatten

In this WT we will be creating a new function that recursively flattens all sub-arrays. Furthermore, we will also demonstrate how to debug your recursive functions, especially in the absence of a debugger, as is our case.

Create a new flow

1. Create a new flow named `mod7.rflatten`
2. Drop a DW to the process area of the flow
3. Turn on the preview

4. Change the output to `application/dw`

The sample data

5. Create a simple array of arrays `[0,1,[2,[3,[4,[5]]]]]`
6. Show that you need four applications of flatten to collapse the subarrays

```
flatten(
      flatten(
            flatten(
                  flatten(
                        [0,1,[2,[3,[4,[5]]]]]
                  )
            )
      )
)
```

The recursive flatten

7. Create a recursive function that just traverses the arrays
   If you are able to traverse, you are able to transform!

```
fun rflatten(a: Array) = a map (
      if (not $ is Array) $ else rflatten($)
)
```

8. Test it using the fore-mentioned array of arrays

```
fun rflatten(a: Array) = a map (
      if (not $ is Array) $ else rflatten($)
)
---
rflatten([0,1,[2,[3,[4,[5]]]]])
```

9. Increment all elements by 1
   To demonstrate that we do indeed traverse every single element let us increment each number by 1

```
fun rflatten(a: Array) = a map (
      if (not $ is Array) $+1 else rflatten($)
)
```

10. Remove the +1 from the function body.
11. Introduce the flatten function to the rflatten body
    a. Where do you think is missing?
    b. Where do you think we should be placing the flatten call?
    c. At which points do we have arrays returned?

```
fun rflatten(a: Array) = flatten(a map (
      if (not $ is Array) $+1 else rflatten($)
))
```

Demonstrate how to debug recursive functions in the absence of a debugger

12. Describe the `log()` function by taking the students in the manual, https://docs.mulesoft.com/mule-runtime/4.2/dw-core-functions-log
13. Copy-n-paste the code in a text editor

```
fun rflatten(a: Array) = flatten(a map (
      if (not $ is Array) $+1 else rflatten($)
))
---
rflatten([0,1,[2,[3,[4,[5]]]]])
```

14. Unravel the first invocation of rflatten

```
rflatten([0,1,[2,[3,[4,[5]]]]])
flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
```

15. Unravel the second invocation of rflatten

```
rflatten([0,1,[2,[3,[4,[5]]]]])
flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
               flatten [2, rflatten([3,[4,[5]]]) ]
```

16. … the third

```
rflatten([0,1,[2,[3,[4,[5]]]]])
flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
               flatten [2, rflatten([3,[4,[5]]]) ]
                           flatten [3, rflatten([4,[5]) ]
```

17. … the fourth

```
rflatten([0,1,[2,[3,[4,[5]]]]])
```

```
flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
               flatten [2, rflatten([3,[4,[5]]]) ]
                           flatten [3, rflatten([4,[5]) ]
                                       flatten [4, rflatten([5])]
```

18. … the last one and get the result

```
rflatten([0,1,[2,[3,[4,[5]]]]])
      flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
                     flatten [2, rflatten([3,[4,[5]]]) ]
                                 flatten [3, rflatten([4,[5]) ]
                                             flatten [4, rflatten([5])]
                                                         flatten [5] = [5]
```

19. Start rolling back the results to the top, one at a time, by replacing the rflatten calls with the results

```
rflatten([0,1,[2,[3,[4,[5]]]]])
      flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
                     flatten [2, rflatten([3,[4,[5]]]) ]
                                 flatten [3, rflatten([4,[5]) ]
                                             flatten [4, [5]] = [4,5]
```

20. Roll back the current last rflatten call

```
rflatten([0,1,[2,[3,[4,[5]]]]])
      flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
                     flatten [2, rflatten([3,[4,[5]]]) ]
                                 flatten [3, [4,5] ] = [3,4,5]
```

21. Roll back the current last rflatten call

```
rflatten([0,1,[2,[3,[4,[5]]]]])
      flatten [ 0,1, rflatten([2,[3,[4,[5]]]]) ]
                     flatten [2, [3, 4, 5] ] = [2,3,4,5]
```

22. Roll back the current last rflatten call

```
rflatten([0,1,[2,[3,[4,[5]]]]])
      flatten [ 0,1,[ 2, 3, 4, 5] ] = [0,1,2,3,4,5]
```

23. The result

```
rflatten([0,1,[2,[3,[4,[5]]]]]) = [0,1,2,3,4,5]
```

24.

Walkthrough 7-3: Traverse and transform the flights objects and sub-objects

In this WT we shall create a set of functions that will allow for traversing any data structure. We will do start by creating a function that is fairly rigid and we shall conclude by providing a more flexible solution.

Back to the flights

1. Head back to `mod6.flights`
2. Open the properties of the DW processor

Define a recursive overloaded function to traverse any data structure

3. Start with Arrays
   For arrays we want to invoke traverse for every single element in the array

```
fun traverse(a: Array) = a map traverse($)
```

4. Continue with Objects
   For objects we want to invoke traverse, once for the key and once for the value

```
fun traverse(o: Object) = o mapObject {(traverse($$)): traverse($)}
```

5. Continue with Keys
   Pay attention to the " ICAO" field name—wouldn't be nice if we can fix such fields such that we trim the empty spaces and upper case all fields?

```
fun traverse(k: Key) = upper(trim(k))
```

6. Continue with Strings
   For strings we only want to lower-case all values, we keep it a little simple for once ☺

```
fun traverse(s: String) = lower(s)
```

7. Apply traverse to the result of the the expression

```
traverse(
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
```

```
) map {
      ($ - "finish"),
      finish: airportInfo[$.finish] -- ["timeZone", "type", "source"]
} map (
      $ reorder (7 to 0)
)
)
```

A more flexible traverse

8. Build a more flexible traverse
    a. Can we do better? How?
    b. How can we make a new traverse function that gives the user more re-usability?
    Please try to create code that is good enough for long enough, the perfect solution is often unattainable.
    We will create a new function that takes another function as input such that we can specify the actions for the simple data types

```
fun traverseFn(a: Array, fn) = a map ( $ traverseFn fn )
fun traverseFn(o: Object, fn) = a mapObject {
      ($$ traverseFn fn): ($ traverseFn fn)
}
fun traverseFn(k: Key, fn) = fn(k)
fun traverseFn(s: String, fn) = fn(s)
```

9. Remove the application of `traverse()`

```
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($ - "finish"),
      finish: airportInfo[$.finish] -- ["timeZone", "type", "source"]
} map (
      $ reorder (7 to 0)
)
```

10. Apply the new `traverseFn` function
    Here we will create an anonymous function to pass to `traverseFn` in order to perform the transformations for the simple data

```
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
      )
) map {
      ($ - "finish"),
      finish: airportInfo[$.finish] -- ["timeZone", "type", "source"]
} map (
      $ reorder (7 to 0)
) traverseFn (
      (e) -> e match {
            case k if (k is Key) -> upper(trim(k))
            case s if (s is String) -> lower(s)
            else -> $
      }
)
```

Cast strings to numbers when feasible

11. Cast the string numbers into the `Number` type
    There are plenty of values in the objects (especially for the airport info) that are numbers, why not just cast them into numbers
    We need to test whether the values are numbers, we can write such a test using regular expressions (regex)—you can test your regex by visiting https://regex101.com/
    You can also just chain another invocation of the `traverseFn` or modify the anonymous function in place, we shall do the latter

```
payload map (
      $ mapObject (
            (v,k,i) -> {(fs2fs[k] default k):v}
```

```
        )
    ) map {
        ($ - "finish"),
        finish: airportInfo[$.finish] -- ["timeZone", "type", "source"]
    } map (
        $ reorder (7 to 0)
    ) traverseFn (
        (e) -> e match {
            case k if (k is Key) -> upper(trim(k))
            case s if (s is String) -> if (
                    s matches /[+|-]?\d+(\.\d*)?/
                ) s as Number else lower(s)
            else -> $
        }
    )
```

Cast to a number and Dates using orElseTry() and orElse().

12. TBC

Walkthrough 7-4: Apply different date formats when casting dates (Optional)

TBC

Walkthrough 7-5: Apply preferences to your data (Optional)

TBC