Dataweave 2.0 Student Manual

September 17, 2020

Contents

Module 1

Fundamentals – Review++

WT 1-1 Import a basic Mule project into Anypoint Studio

Create new project

1. Create a new project and call it `dataweave`
2. Copy the `examples/airport-info.csv` to `dataweave/src/main/resources`
3. Copy the `examples/flights-example.xml` to `dataweave/src/test/resources`
4. Copy the `examples/flight-example.json` to `dataweave/src/test/resources`

WT 1-2 Fundamentals review++

In this WT the goal is to attempt (I am saying attempt because often enough we have participants who don't meet the prerequisites) 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++`
   The reason for prefixing the flow name with the name of the flow is one of good practice. Such a convention will improve the readability of your flows by identifying the Mule Configuration file a flow is defined under by just looking at a Flow Reference's display name.
3. Drop a DW (aka `Transform Message`) processor 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. Create an empty object

```
%dw 2.0
output application/java
---
{}
```

DW is JSON-like, that is the designers of DW borrowed syntax and semantics from JSON. As such, the semantics of in DW is object creation.

9. Create an empty array

```
%dw 2.0
output application/java
---
[]
```

Fields

10. Use the dot notation to access the `airline` field

```
%dw 2.0
output application/java
---
payload.airline
```

11. Use the bracket notation to access the `airline` field

```
%dw 2.0
output application/java
---
payload["airline"]
```

Dynamically evaluating field names and accessing them can easily be achieved using this bracket notation.

12. Use the index 0 to access the `airline` field

```
%dw 2.0
output application/java
---
payload[0]
```

Let me let you in a secret: Objects internally are represented as arrays—-field access is a facade

13. Create an object with repeating field names

```
%dw 2.0
output application/java
---
{
  flight: 1,
  flight: 2,
  flight: 3
}
```

Here lies the explanation as to why DW objects are stored internally as arrays. An object containing duplicate field names can only access any of these duplicates fields other than the first through indexes.

14. Transform data into XML

```
%dw 2.0
output application/xml
---
flights: {
  flight: 1,
  flight: 2,
  flight: 3
}
```

Because DW must support XML and because XML has no clue of collections other than repeating tags, how else do we expect to be able to generate XML with tags that repeat in DW if not for fields repeating inside DW objects. It is ingenious!

String concatenation

15. Change the `output` to `application/dw`
16. Concatenate strings using the ++ function

```
%dw 2.0
output application/dw
---
"The flight is operated by " ++ payload.airline
```

Setting the `output` to `application/dw` should ONLY be done while statically testing in order to get additional information from the DW compiler/interpreter–i.e. we are debugging our code through Studio's preview. Using `application/dw` will cause signifcant delays during runtime.

17. Concatenate strings using the `$()` string sub-expression

```
%dw 2.0
output application/dw
---
"The flight is operated by $(payload.airline)"
```

We have to ensure the expression inside `$()` evaluates to a string or it is automatecally casted into a string, otherwise we will be getting type missmatch errors.

Conditional expressions

18. `if then else` conditional

```
%dw 2.0
output application/dw
---
if (true) 1 else 0
```

DW being a functional programming language does not have any statements, as such this is a conditional expression that must evaluate to a value.

19. Nullity conditional

```
%dw 2.0
output application/dw
---
payload.airline default "Delta"
```

This is a shortcut to the `if (payload.airline == null) "Delta" else payload.airline`

20. Conditional elements for objects

```
%dw 2.0
output application/dw
---
{
  (a: 1) if (true),
  (b: 2) if (false),
  c: 3
}
```

21. Conditional elements for arrays

```
%dw 2.0
output application/dw
---
[
  (1) if (true),
  (2) if (false),
  3
]
```

Array and String access and Ranges

22. Retrieve the first element of an array

```
%dw 2.0
output application/dw
---
[2,6,4,1,7][0]
```

23. Retrieve the last element of an array

```
%dw 2.0
output application/dw
---
[2,6,4,1,7][-1]
```

24. Retrieve a subset of an array

```
%dw 2.0
output application/dw
---
[2,6,4,1,7][1 to -2]
```

25. Reverse the elements of an array

```
%dw 2.0
output application/dw
---
[2,6,4,1,7][-1 to 0]
```

26. Retrieve the first character of string

```
%dw 2.0
output application/dw
---
payload.airline[0]
```

27. Retrieve the last character of a string

```
%dw 2.0
output application/dw
---
payload.airline[-1]
```

28. Retrieve a substring

```
%dw 2.0
output application/dw
---
payload.airline[0 to 2]
```

29. Reverse a string

```
%dw 2.0
output application/dw
---
payload.airline[-1 to 0]
```

Common functions and operators

30. `typeOf`

```
%dw 2.0
output application/dw
---
typeOf({})
```

This is a great function for debugging–again and it will help us identify the types of data we are working with. We will use it a few times to gain clarity when all else has failed.
The documentation for this function can be see here.

31. `sizeOf`

```
%dw 2.0
output application/dw
---
sizeOf([2,6,4,1,7])
```

The documentation for this function can be see here.

32. `contains`

```
%dw 2.0
output application/dw
---
[2,6,4,1,7] contains 2
```

The documentation for this function can be see here.

33. `is`

```
%dw 2.0
output application/dw
---
{} is Object
```

The `is` operator is for testing type membership.

Expression chaining

34. Iterate the `[2,5,3,7,8]` and increment each number by one

```
%dw 2.0
output application/dw
---
[2,5,3,7,8] map $+1
```

35. Chain another expression and substract each number by one

```
%dw 2.0
output application/dw
---
[2,5,3,7,8] map $+1 map $-1
```

Because DW supports infix function invocation for all functions with exactly two arguments, we are able to chain one function after the other making our code more readable.

Transform XML to JSON

36. Create a new flow and name it `mod1-xml2json`
37. Set the input payload metadata to `src/test/resources/flights-example.xml`, name the new type `flights_xml`
38. Edit the sample data
39. Turn on the preview
40. change the output to `JSON`
41. Replace `{}` to `payload`
42. Explore the structure in the Preview and focus on the objects created with `return` fields repeating
43. Is this a valid JSON data structure?
    According to the JSON specification this is a valid JSON. But it is not appropriate.
    Transform the XML into a JSON collection containing the objects found under `return` tags
44. Use the `..*` selector to perform a recursive search and find fields named `return`

```
payload..*return
```

45. Go to the first element in the sample data under the `return` tag
46. 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>
```

47. Illustrate that `..*` performs a breadth-first search and the output contains an extra result all the way to the bottom.
48. Use the `.*` selector to perform a search at the right level–no longer do we receive the next `return` result.

```
payload.findflightResponse.*return
```

49. Restore your sample data by removing the additional nested `<return>10</return>` XML tag.
50. Ensure you make use of the namespace from the input data. Ignoring namespaces is not advised unless you are certain the data will always look the same, you will never have another `findFlightResponse` tag with a different meaning

```
ns ns2 http://soap.training.mulesoft.com/
---
payload.ns2#findFlightResponse.*return
```

51. Copy all the data from the preview
52. Create a new file under `src/test/resources` and call it `flights.json`

Transform JSON to XML

53. Create a new flow and name it `mod1-json2xml`
54. Drop a DW to the process area
55. Set the input payload metadata to `src/test/resources/flights.json`
56. Edit the sample data
57. Turn on the preview
58. Change the output to XML
59. Replace `{}` to `payload`
60. The error error says `Cannot coarse an array ... to a String`
    (a) 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 and serialization of the array type.
    (b) We need to proceed by eliminating the arrays
61. Create an appropriate XML for just two elements of the inputs

```
%dw 2.0
output application/xml
---
flights: {
  flight: payload[0],
  flight: payload[1]
}
```

62. Set the output to `application/dw` and identify the internal data structure we must aim for when generating XML
63. We need to generate XML for all elements not just the first two, change the code so that we now iterate over the collection of data in the `payload`

```
%dw 2.0
output application/dw
---
flights: payload map {
  flight: $
}
```

64. Switch the output back to XML results in errors because we are still having an array in our data structure

65. Change the output yet again to `application/dw`

66. Eliminate the array by enclosing the `map` in $\{()\}$

```
%dw 2.0
output application/dw
---
flights: {(payload map {
  flight: $
})}
```

The semantics of $()$ are the usual precedence operators, however ***the semantics of parenthesis change when they appear on their own within $\{\}$ enclosing (i) objects or (ii) 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.*

67. *So far we solved this transformation by following a top-to-bottom solution. You can also solve this transformation by following a bottom-up approach.*
Change the expression back to just `payload` and eliminate the array first!

```
%dw 2.0
output application/dw
---
flights: {(payload)}
```

68. Organize the records around their own tag before we destroy the array and collapse the first level of containing objects.

```
%dw 2.0
output application/dw
---
flights: {(payload map flight: $)}
```

Note that objects with a single field can have the $\{\}$ omitted

69. Finally, change the output back to XML

Module 2

Use Case: XML flights to JSON

As a developer I want to receive XML flight records and generate an enhanced JSON data structure with the corresponding flights.
The JSON data structure is enhanced as follows:

- It contains the corresponding appropriate JSON structure contain the flights from the XML inputs
- It is enhanced with total seating capacity per flight
- It is enhanced with prices adjusted across a number of currencies
- It is enhanced such that important KPIs are contained at the same level as the flights

WT 2-1 Organize DataWeave code with variables and functions

Create a new flow

1. Create a new Mule Configuration file and name it `mod2`, it will contain the solutions to all WTs from module 2.
2. Create a new flow named `mod2-flights`
3. Drop a DW to the process area of the flow
4. Define the payload input metadata to `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

```
%dw 2.0
output application/json
---
payload..*return
```

Create a variable

9. Create a variable visible throughout the DW expression

```
var theTotalSeats = 400
```

10. Add the `totalSeats` field to the existing list of objects, do it for a single object then do it for all objects in the collection

```
%dw 2.0
output application/json
var theTotalSeats = 400
---
payload..*return[0] ++ {
  totalSeats: theTotalSeats
}
```

11. Do it now for all elements

```
%dw 2.0
output application/json
var theTotalSeats = 400
---
payload..*return map ($ ++ {
  totalSeats: theTotalSeats
})
```

++ we have already seen when concatenating strings we see it operating with objects as well because it is overloaded, more on overloading soon.

12. There is another way to add a field(s) to an existing object

```
%dw 2.0
output application/json
var theTotalSeats = 400
---
payload..*return map {
  ($)
  totalSeats: theTotalSeats
}
```

We have already seen $\{()\}$ when eliminating arrays, here these $()$ are applied to single objects with the same effect; i.e. destroy the object and retrieve the basic building blocks of the object, that is the keys and the associated values. These basic building blocks are then introduced in the new object created by the outermost object.
Pick the method you prefer to concatenate objects, I prefer the latter which is the one I shall be using for the duration of this class.

Fix the Boing typo

13. Add another field containing the `planeType` value

```
%dw 2.0
output application/dw
var theTotalSeats=400
---
payload..*return map {
  ($),
  planeType: $.planeType,
  totalSeats: theTotalSeats
}
```

A quick inspection of the output identifies two `planeType` fields containing the same value. This is a problem because we are interested in a single field containing the corrected spelling for Boeing. We should remove one of them and removing the first one is the right choice because this first one is the one we have no control over. The second `planeType` is the one we have full control because we are adding it!

14. Remove the first `planeType` field

```
%dw 2.0
output application/dw
var theTotalSeats=400
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType,
  totalSeats: theTotalSeats
}
```

This unary – operator is overloaded to also apply to objects and remove fields. You can keep applying it successively if you want to remove multiple fields–e.g. `$ - "planeType" - "price"`

15. Fix the `Boing` typo

```
%dw 2.0
output application/dw
var theTotalSeats=400
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: theTotalSeats
}
```

It is worth noting that `replace` and `with` are two separate functions where the result of `replace` is the first argument to `with`. It makes the code "talk to you in English"! This is the beauty of infix function invocation.

Calculate the total seats as a function of the **planeType** using **fun**

16. Create and apply a function and start unit-testing it

```
%dw 2.0
output application/dw
var theTotalSeats = 400
fun getTotalSeats(pt) = pt
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeats($.planeType)
}
```

`pt` is a user defined arbitrary name, denoting the sole input parameter
By unit testing we refer to the method by which small chunks of our functionality is tested before we put them all together. We are not referring to automated/regression testing.

17. Create the condition that identifies 737s over the other types of planes

```
%dw 2.0
output application/dw
var theTotalSeats = 400
fun getTotalSeats(pt) = pt contains "737"
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeats($.planeType)
}
```

18. Enclose the condition in an `if` expression

```
fun getTotalSeats(pt) = if (
  pt contains "737"
) 150 else 300
---
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeats($.planeType)
}
```

19. 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 "707" or
  pt contains "727"
) 150 else 300
```

20. Fix the error `Cannot coerce String (737) to Boolean`

```
fun getTotalSeats(pt) = if (
  (pt contains "737") or
  (pt contains "707") or
  (pt contains "727")
) 150 else 300
```

Parenthesizing to enforce precedence is required in this context because `or` has higher precedence vs `contains` A chunk of the issues you will have when you start writing DW expressions on your own will stem from precedence rules.

21. Discuss issues with the `getTotalSeats` functions
    (a) We execute this function once per record
    (b) We are searching strings
    (c) We do this string search three times
    (d) The function is not that efficient, we could do better

Please do not think for a moment that in modern computing string searches as slow, they are fast and could be optimized in a number of ways. Nonetheless, this discussion has merit in the presence of large to very large data sets where the function is called once per record; i.e. every little bit helps!

Calculate efficiently the total seats as a function of **`planeType`** using a lambda expression

22. Create another function named `getTotalSeatsL`

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

L stands for Lambda, we store an anonymous function to a variable; i.e. we provide this anonymous function with a name. Additionally, the body of this function evaluates into the argument we passed–this encourages unit testing.

23. Apply the function to in the expression and get results

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType)
}
```

Applying the function as soon as possible and getting results as we further develop our function will only facilitate unit testing and code correctness.

24. Get the last three characters of the plane type

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

We can now use the number to compare instead of doing a string search which will speed up the execution of our code.

25. Introduce closures (i.e. localized declarations) using `do {}`

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

A closure is a construct that allows for the declaration of variables, functions, etc with a localized scope. The `--` serve the same purpose like the `--` we see in other DW expressions, they are section separators used to separate the declarations and the expression.

26. Add the conditional to the function

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

You can use either one of these two functions to calculate the total seats; however, if you would like to use features such as function overloading you MUST stick with the `fun`.

Adjust price for currency

27. Create an object that contains currency exchange rates

```
var xes = {
  USD: 1.0,
  EUR: 0.9,
  GBP: 0.8,
  CAD: 1.3,
  AUD: 1.5,
  MXN: 25,
  INR: 72
}
```

We hard-code these currencies because we are within the confines of training. We can easily fetch these currencies dynamically from any data source and generate the map.

28. Create a function to calculate the price for a currency

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

29. Apply the function in prefix syntax

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price,"EUR")
}
```

30. Apply the function in infix syntax

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price,"EUR"),
  priceGBP: $.price adjustFor "GBP"
}
```

Functions with exactly two arguments get this infix application support! In fact, infix function application is encouraged because (1) it is natural in its application, (2) no need to use excessive parenthesis, and (3) allows for a more natural application of expression chains.

WT 2-2 Reuse DataWeave transformations

Store DW code in a file

1. Switch to the XML view of your file
2. Navigate under the `mod2-flights` 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 `mod2-flights`
5. Click the Edit current target button (pencil icon)
6. Click the radio button File and type `dw/transforms/mod2/flights` in the text field to the right
7. Click OK *From the point of view of the DW properties UI nothing has changed. Nonetheless, with this action we have stored the DW code inside a new file under `src/main/resources/dw/transforms/mod2` named `flights.dwl`*

Reuse the DW code from the file

8. Create a new flow named `mod2-reuse-flights`
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. Introduce the / closing to the opening `<ee:set-payload />` tag, this should automatically remove the explicit closing tag
14. Add the attribute `resource` to the `<ee:set-payload />` tag

```
<ee:set-payload resource="dw/transforms/mod2/flights.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 attempt to reuse the file, you would be overwriting it! That pencil button is a "one way trip", only there to store the file not reference it.

15. Switch back to the graphical view
16. Open the properties of the the DW processor under the `mod2-reuse-flights`
17. Turn on the Preview *There is an issue that indicates that there is no metadata identifying what the payload is. You can fix this issue by just setting the metadata again. This issue is displayed because of DataSense. This issue is only visible when in Studio, if we start the server and deploy our app, DataSense is never in play.*
18. Set the input payload metadata to `flights_xml`
19. Turn on the Preview and validate you see the result

WT 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/modules`
3. Create a new file under `dw.modules` and name it `Currency.dwl`
4. Type on line 1

```
%dw 2.0
```

DW modules can only contain declarations. Declarations such as variables, function, types, etc.

5. Navigate back to the DW processor under `mod2-reuse-flights`
6. Copy the `xes` variable and the `adjustFor` function
7. Paste to `Currency.dwl` under line 1 and save

Use the module

8. Go back to the DW processor under `mod2-reuse-flights`
9. Use the module by fully qualifying the function to adjust the price for the CAD currency

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD"
}
```

10. Import the new module below the output directive

```
import dw::modules::Currency
```

11. Use the module again this time by taking advantage of the `import` to adjust the price for AUD

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD"
}
```

12. Import the module again and provide an an alias to the module

```
import dw::modules::Currency as Curr
```

13. Use the module through the `Curr` alias next

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD",
  priceMXN: $.price Curr::adjustFor "MXN"
}
```

14. Import all declarations to the current namespace

```
import * from dw::modules::Currency
```

15. Use directly the `adjustFor` function

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD",
  priceMXN: $.price Curr::adjustFor "MXN",
  priceINR: $.price adjustFor "INR"
}
```

There is also an inline version of the `adjustFor` function which takes precedence. As such we are not using the function provided by the module. We can very easily verify by changing the body of the inline function, just change the body to 1000.

16. Modify the last `import` to selectively import declarations and provide them with aliases

```
import adjustFor as adj4 from dw::modules::Currency
```

17. Use the `adj4` alias

```
payload..*return map {
  ($ - "planeType"),
  planeType: $.planeType replace /Boing/ with "Boeing",
  totalSeats: getTotalSeatsL($.planeType),
  priceEUR: adjustFor($.price, "EUR"),
  priceGBP: $.price adjustFor "GBP",
  priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
  priceAUD: $.price Currency::adjustFor "AUD",
  priceMXN: $.price Curr::adjustFor "MXN",
  priceINR: $.price adj4 "INR"
}
```

WT 2-4 Optional: Calculate KPIs using the Arrays Module

Inspect the Arrays module in the Documentation

1. Navigate to the Arrays module page in the documentation
2. Visit the `countBy` function documentation and understand the signature
3. Visit the `every` function documentation and understand the signature
4. Visit the `some` function documentation and understand the signature
5. Visit the `sumBy` function documentation and understand the signature

Add the KPI section

6. Add an object and set the record of flights to the object's `data` field

```
{
  data: payload..*return map {
    ($ - "planeType"),
    planeType: $.planeType replace /Boing/ with "Boeing",
    totalSeats: getPlaneTypeL($.planeType),
    priceEUR: adjustFor($.price,"EUR"),
    priceGBP: $.price adjustFor "GBP",
    priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
    priceAUD: $.price Currency::adjustFor "AUD",
    priceMXN: $.price Curr::adjustFor "MXN",
    priceINR: $.price adj4 "INR"
  }

}
```

7. Add an `kpi` field above the `data` field and assign an empty object to it

```
{
  kpi: {

  },
  data: payload..*return map {
    ($ - "planeType"),
    planeType: $.planeType replace /Boing/ with "Boeing",
    totalSeats: getPlaneTypeL($.planeType),
    priceEUR: adjustFor($.price,"EUR"),
    priceGBP: $.price adjustFor "GBP",
    priceCAD: $.price dw::modules::Currency::adjustFor "CAD",
    priceAUD: $.price Currency::adjustFor "AUD",
```

```
    priceMXN: $.price Curr::adjustFor "MXN",
    priceINR: $.price adj4 "INR"
  }
}
```

We will use this new field to store the calculated KPIs in subsequent steps of this WT.

Assert all flights are Delta operated

8. Import the `Arrays` module

```
import every,some,countBy,sumBy from dw::core::Arrays
```

9. Use the `every` function to assert whether all flights are Delta operated

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta"
},
```

You may want to refactor your code so that you execute `payload..*return` once. You already have seen how to do it, consider it a bonus exercise.

Assert the existence of full flights

10. Use the `some` to assert whether there are any full flights

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..*return some ($.emptySeats == 0)
},
```

The result is `false`, i.e. there are no full flights. However, the second to last flight is a full flight because the `emptySeats` tag is set to 0. The problem lies with the comparison–we are comparing a string with a number. You can either use a type cast to `$.emptySeats as Number == 0` or the equality operator `~=` that autocasts its arguments.

11. Use the operator `~=` to fix the `Boolean` condition

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..*return some ($.emptySeats ~= 0)
},
```

Calculate the number of full flights

12. Calculate the number of full flights

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..*return some ($.emptySeats ~= 0),
  fullFlightsCount: payload..*return countBy ($.emptySeats ~= 0)
},
```

Sum the total number of empty seats across all flights

13. Sum up all the `emptySeats` across all flights

```
kpi: {
  allDeltaFlights: payload..*return every (e) -> e.airlineName == "Delta",
  anyFullFlights: payload..*return some ($.emptySeats ~= 0),
  fullFlightsCount: payload..*return countBy ($.emptySeats ~= 0),
  totalEmptySeats: payload..*return sumBy $.emptySeats
},
```

We should be seeing errors now. These errors are false-positives due to DataSense. You could potentially eliminate them by adding `payload..*return default []` to the right of every single `Array` function.

Module 3

Defensive programming

WT 3-1 Function Overloading

Create a new flow

1. Create a new Mule Configuration file and name it `mod3`
2. Create a new flow named `mod3-data-matcher`
3. Drop a DW to the process area of the flow
4. Turn on the preview
5. Switch to the Source Only view
6. Change the output to `application/dw`

Nullity and function signatures

7. Explore nullity

```
%dw 2.0
output application/dw
---
typeOf(null)
```

By looking at the preview we can quickly realize that DW's typing system separates nullity from all other types. The benefit of such separation is that you can catch a good chunk of nullity errors at compile time.

8. Replace the expression with just `sizeOf`
9. Examine the signature in the preview

```
(arg0:Array | Object | Binary | String) -> ???
```

`sizeOf` expects a single argument. This argument can be one of listed types separated by a pipe symbol (|). This | indicates that this function has been overloaded.
Only through `application/dw` can we see the signature of functions.

10. Replace the expression with just `map`
11. Examine the signature in the preview

```
(arg0:Array | Null, arg1:Function | Function) -> ???
```

`map` is also an overloaded function, but this time it is a function that takes two arguments. If we were to reconstruct these overloaded functions defining `map` then we can tell (1) that there are two overloaded functions because we have a single pipe separating the types of the arguments and (2) the arguments of these overloaded functions will look as follows by using the position of the corresponding arguments:

- ***map*(*Array*,*Function*) – *i.e. `Array` will match with the first `Function`***
- ***map*(*Null*, *Function*) – *i.e. `Null` will match with the second `Function`***

Overload a function

12. Create a function that accepts the `Null` type

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
---
dataMatcher(null)
```

The `:` separates the name of the argument with the type of the argument. The same type of syntax can be used to specify types when needed.
You have created a function that takes only null values.

13. Pass an empty array instead

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
---
dataMatcher([])
```

You are now getting errors because our function is not defined to accept arrays.

14. Overload the function to also accept arrays

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
fun dataMatcher(a: Array) = "Array found"
---
dataMatcher([])
```

*Only a function declared through **fun** can be overloaded–in other words you cannot overload λ-expressions*

15. Display the signature of the `dataMatcher` function

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
fun dataMatcher(a: Array) = "Array found"
---
dataMatcher
```

16. Examine the signature

```
(arg0:Null | Array) -> ???
```

The signature clearly show the types the function accepts.

17. Overload the function for **Object**, **Number**, and **String**

```
%dw 2.0
output application/dw
fun dataMatcher(n: Null) = "Null found"
fun dataMatcher(a: Array) = "Array found"
fun dataMatcher(o: Object) = "Object found"
fun dataMatcher(n: Number) = "Number found"
fun dataMatcher(s: String) = "String found"
---
dataMatcher
```

18. Examine the signature one last time

```
(arg0:Null | Array | Object | Number | String) -> ???
```

WT 3-2 Pattern Matching

Matching literals

1. Stay under the `mod3-data-matcher` flow
2. Delete the current expression
3. Use the `match` operator to match `Null` values

```
null match {
  case null -> "Null found"
}
```

This is the `match` operator, not to be confused with the `match` function.

4. Match the number 10

```
10 match {
  case null -> "Null found"
  case 10 -> "Ten"
}
```

Cases are evaluated in order of appearance, ensure more general cases are below the more specific ones.

5. Match the string “ABC”

```
"ABC" match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
}
```

6. Match the empty array

```
[] match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
}
```

7. Match the empty object

```
{} match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
  case {} -> "Empty object found"
}
```

8. Add elements to the object

```
{a: 1} match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
  case {} -> "Empty object found"
}
```

We are getting errors because there is no case matching this expression.
There is no way to match inner data of objects and arrays directly in a case–i.e. adding inner data at the case level will only result in syntax errors.

9. Add the default case for when there is no case matching your data

```
{a: 1} match {
  case null -> "Null found"
  case 10 -> "Ten"
  case "ABC" -> "Alphabet"
  case [] -> "Empty array found"
  case {} -> "Empty object found"
  else -> {message: "No match", data: $}
}
```

In the body of the `else` case we are back-referencing our data with a single `$`. We can back-reference our data in all other cases in the same way. Further along we shall see how to define our own placeholder to back-reference our data so that we don't depend on a single `$`

Matching types

Lets use pattern matching to replicate function overloading from before.

10. Remove or comment-out the expression
11. Match the `Null` type

```
null match {
  case is Null -> "Null found"
  else -> $
}
```

*The `is` operator is used in the first case to match all values of type **Null**.*
We pro-actively set the default case to just evaluate back to the expression we are matching.

12. Match **Number**, **String**, **Object**, and **Array** types

```
[] match {
  case is Null -> {message: "Null found", data: $}
  case is Number -> {message: "Number found", data: $}
  case is String -> {message: "String found", data: $}
  case is Object -> {message: "Object found", data: $}
  case is Array -> {message: "Array found", data: $}
  else -> {message: "No match", data: $}
}
```

We are back-referencing the values we are matching using the $ for all our cases.

Matching by any condition

13. Replace the $ with a placeholder of our own choosing

```
[1] match {
  case n if (n is Null) -> {message: "Null found", data: n}
  case n if (n is Number) -> {message: "Number found", data: n}
  case s if (s is String) -> {message: "String found", data: s}
  case o if (o is Object) -> {message: "Object found", data: o}
  case a if (a is Array) -> {message: "Array found", data: a}
  else other -> {message: "No match", data: other}
}
```

These new placeholders are arbitrary, they can be anything of your choosing, from single letter to words–very similar to declaring variables. Using this new syntax to define our placeholders and conditionals opens an unlimited set of Boolean expressions that could be defined on a per case basis.

14. Use pattern matching to distinguish the empty array, the array that has less than or equal to 100 elements, and the array that has larger than 100 elements

```
(0 to 200) match {
  case [] -> "Empty Array"
  case a if (a is Array and sizeOf(a) <= 100) -> "Non-empty array with less or equal to 100 elements"
  else a -> "Non-empty array with larger than 100 elements"
}
```

For this use case we combine all of the above:

- *Literal pattern matching for the empty array*
- *Conditional pattern matching on the type and size*
- *The default case for all other arrays*

WT 3-3 Error Handling

Create a new flow

1. Create a new Mule Configuration file and name it `mod3`
2. Create a new flow named `mod3-errors`
3. Drop a DW to the process area of the flow
4. Turn on the preview
5. Switch to the Source Only view
6. Change the output to `application/dw`

The error 10 / 0

7. Introduce an error

```
%dw 2.0
output application/dw
---
10 / 0
```

When the evaluation of DW expressions result in errors the whole expression errors out–there are no partial results
Imagine scenarios where we are working with sequences or records where some of these records our malformed, DW expressions will not provide you with partial results, instead the full expression will error out. Error handling will allow for such partial results.

The `dw::Runtime::try` function

8. Navigate to the Dataweave documentation

9. Examine the signature of the dw::Runtime::try function found in the Dataweave Reference section
The `dw::Runtime::try` function takes a single argument. The argument is a λ-expression that takes no arguments itself. The body of this λ-expression is the expression that we want to evaluate for errors.
This λ-expression is also referenced as a delegate function because through it we are delegating the execution of an expression to `dw::Runtime::` function. The latter will determine whether our expression has errors or not.
The return value is a `TryResult<T>` structure. Thankfully, we have documentation that describes this structure.

10. Test the `10 / 0` expression

```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 0)
```

11. Examine the result

```
{
  success: false,
  error: {
    kind: "DivisionByZeroException",
    message: "Division by zero",
    location: "\n4| dw::Runtime::try(() -> 10 / 0)\n                                ^^^^^^",
    stack: [
      "main (anonymous:4:24)"
    ]
  }
}
```

We no longer have errors preventing us from getting results in the preview. Instead we are looking at a `TryResult` structure where the `success` field is set to `false` indicating that our expression has errors. Finally the error's description can be seen in the `error` field.

12. Test the `10 / 2` expression

```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 2)
```

13. Examine the result

```
{
  success: true,
  result: 5.0
}
```

Similarly we have `success` set to `true` but instead of `error` we have the `result` field set to the value our expression evaluates to.

The `guard` function

14. Chain the `match` operator

```
%dw 2.0
output application/dw
---
dw::Runtime::try(() -> 10 / 0) match {
  case tr if (tr.success) -> tr.result
  else tr -> tr.error.message
}
```

When `success` is set to `true` just return the `result` otherwise arbitrarily just return the `error.message`

15. Extrapolate to a function that guards against errors

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

There is no need for a `guard` function. There are additional error handling functions that we shall see further along the material that provide for complete and flexible error handling. Nonetheless, going through the refactoring of our code into a function allows us to take a peek on functional programming.

16. Test the `guard` function with a correct expression

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

17. Test the `guard` function with an erroneous expression

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

WT 3-4 Optional: 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

List of String Dates

1. Stay in the `mod3-errors` flow
2. Create an array of string dates and assigned it in a variable

```
var dates = [
  "2020/01/01",
  "20100101",
  "2000/01/01"
]
```

It is the second date that is malformed and it is the cause of failure of our transformation.

3. Remove or comment out the current expression
4. Iterate over dates and cast into a `Date` type

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

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

Using `guard` and obtaining partial results

5. Delegate the execution of the type-casting expression to the `guard` function

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

6. Finally `filter` for `Date` types and get your partial results while ignoring the rest

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

7. We can expand on this solution such that we also obtain the errors

```
do {
  var parsedDates = dates map (
  guard( () -> $ as Date {format: "yyyy/MM/dd"} )
  )
  ---
  {
    success: parsedDates filter ($ is Date),
    failure: parsedDates filter  not ($ is Date)
  }
}
```

Note to instructors: This is a good exercise to have students work on their own for "bonus points", after all we have already covered all topics they need to know to get it done.

Module 4

Use Case: Flights and Airports

In this module we work on the main use-case of the class. This use-case entails combining flights with airports and transforming the data such that are communicated with external systems. The tasks we shall perform are as follows:

- Dynamically rename fields
- Combine data from two sources
- Explore in more detail functional programming
- Optimize your code in the absence of profiling tools
- Reorder objects to meet legacy system criteria
- Traverse and transform any type of structured data

In fact, the remainder of the class makes use of this use-case. We shall still apply unit-testing to solve smaller issues before we integrate the unit-tested expressions in our main transformation.

WT 4-1 Change field names

Create a new flow

1. Create a new Mule Configuration file and name it `mod4`
2. Create a new flow named `mod4-flights-airports`
3. Drop a DW to the process area of the flow
4. Define the payload input metadata to `flights_json`
5. Edit the sample data
6. Turn on the preview
7. Change the output to `application/dw`

Create the map

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

```
var fs2rn = {
  airlineName: "carrier",
  departureDate: "date",
  emptySeats: "seats",
  planeType: "plane"
}
```

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 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

9. Access the first object

```
%dw 2.0
output application/dw

var fs2rn = {
  airlineName: "carrier",
  departureDate: "date",
  emptySeats: "seats",
  planeType: "plane"
}
---
payload[0]
```

10. Visit the `mapObject` documentation page and explore its signature. Focus on the return type of the λ-expression.
11. Iterate over the object and return the empty object form the λ-expression

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

Notice the result in the preview, it is also an empty object. `mapObject` invokes the λ-expression once for every single pair of key and value in order of appearance. The result of `mapObject` is the concatenated objects returned from the λ-expression.

12. Reform the object

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

13. Examine the preview

```
{
  k: "Delta",
  k: "A1B2C3",
  k: "2018/03/20",
  k: "SFO",
  k: "40",
  k: "MUA",
  k: "Boing 737",
  k: "400.0"
}
```

Notice that the values are being reformed, that is the `v` is evaluated, while the keys, `k`, is displayed verbatim.

14. Force the evaluation of the field name

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

We must enclose the key inside `()` to force its evaluation. It is how the language works to facilitate the creation of fields–on the one hand we don't have to make excessive use of quotes enclosing the hard-coded fields while on the other we must enclose the keys inside parenthesis when dynamically evaluating them.
It is worth the effort to inspect the types of fields when using the `mapObject` function. By enclosing the `(sizeOf(k))` you will see in the preview that fields are of a specific type `Key`. This is a critical detail that we will latch on when working on the final exercise of the class.

Change the field names

15. Dynamically evaluate the key based upon the `fs2rn` map for a single object

```
payload[0] mapObject (v,k,i) -> {(fs2rn[k]): v}
```

16. Examine the error: `Cannot coerce Null (null) to Key`
The error is due to keys not appearing in the `fs2rn` map. For example, `code`, `price`, etc.

17. Accommodate for unchanged keys

```
payload[0] mapObject (v,k,i) -> {(fs2rn[k] default k): v}
```

18. Change the field names for all objects in the array

```
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
```

WT 4-2 Combine flights and airports

Explore the CSV file

1. Navigate under `src/main/resources` folder
2. Right click the `airport-info.csv`
3. Select `Open With` and then click `Text Editor`
4. Discuss the contents and identify bad data
Dealing with bad data now days is the norm, as such this CSV file contains bad data. There are two instances of such bad data:
    - *In the header the `ICAO` header has an empty space in the prefix*
    - *Rows 7 and 13 are duplicated when this CSV file should contain unique records*

Parse the CSV airports

5. Navigate back to `mod4` mule configuration file
6. Create a new flow named `mod4-read-parse-airports`
7. Drop a DW to the process area of the flow
8. Turn on the preview
9. Change the output to `application/dw`
10. Just type the function `readUrl` in the body of the expression

```
%dw 2.0
output application/dw
---
readUrl
```

11. Explore the signature

```
(first:String | Binary, second:String, third:Object) -> ???
```

The details of `readUrl` function can be found in this page.

12. Read and parse the file

```
%dw 2.0
output application/dw
---
readUrl(
  "classpath://airport-info.csv",
  "application/csv",
  {
    separator: ","
  }
)
```

The last argument, aka reader properties, is not needed because the default separator is the , symbol. It is only here for illustration. You can find all reader properties, along with the default values, in this documentation page.

13. Fix the duplicate records

```
%dw 2.0
output application/dw
---
readUrl(
  "classpath://airport-info.csv",
  "application/csv",
  {
    separator: ","
  }
)
distinctBy $.IATA
```

IATA stands for International Air Transport Association and its values are standardized across the industry to be unique codes identifying airports. As such we make use of just this value to create a set out the collection–i.e. get the unique values.

14. Incorporate the code in the `mod4-flights-airports` flow

```
%dw 2.0
output application/dw

var fs2rn = {
  airlineName: "carrier",
  departureDate: "date",
  emptySeats: "seats",
  planeType: "plane"
}

var airports = readUrl(
  "classpath://airport-info.csv",
  "application/csv",
  {
    separator: ","
  }
)
distinctBy $.IATA

---
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
```

Inject the airport to each flight

15. Chain another `map` function

```
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {

}
```

There is no need for another chained `map` function, can can embed the combination of the data in a single iteration. However, since we are in training clarity takes precedence vs performance.

16. Reintroduce the fields from the previous map

```
map {
  ($)
}
```

17. Add the `airport` field and assign to it all airports

```
map {
  ($),
  airport: airports
}
```

18. Nest `filter` to search for the `SFO` airport

```
map {
  ($),
  airport: airports filter ($.IATA == "SFO")
}
```

19. Dynamically make use of the flight's `destination` field to search the airports

```
map {
  ($),
  airport: airports filter ($.IATA == $.destination)
}
```

20. Examine the output in the preview

```
[
  {
    carrier: "Delta",
    code: "A1B2C3",
    date: "2018/03/20",
    destination: "SFO",
    seats: "40",
    origin: "MUA",
    plane: "Boing 737",
    price: "400.0",
    airport: []
  },
  ...
]
```

Notice the empty data when earlier we were displaying the SFO record. The reason for this is because we have nested a filter right inside a map function and we choose to make use of the $ placeholders in both cases!

21. Change `filter`'s λ-expression argument names fixing the transformation

```
map {
  ($),
  airport: airports filter  (e) -> (e.IATA == $.destination)
}
```

Not only we fixed our transformation by combining each flight with an airport record we also fixed the airport record that is associated with a flight to match the flight's record `destination` field value.

Functions as values

22. Create and apply a function that does the filtering

```
var filterAirportsByIATA = (dest) -> airports filter ($.IATA == dest)
---
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
  airport: filterAirportsByIATA($.destination)
}
```

For the most part the name of this functions gives away that every single aspect is hard-coded, apart from the IATA code passed to the `dest` argument.

23. Create and apply a "naive" generic filter function

```
var filterAirportsByIATA = (dest) -> airports filter ($.IATA == dest)
var genericFilter = (a,f,v) -> a filter ($[f] == v)
---
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
  airport: genericFilter(airports,"IATA",$.destination)
}
```

This is a naive version of a generic filter because we have hard-coded the condition! A generic filter function should be able to take any kind of a `Boolean` condition and filter records of data.

24. Rename the `genericFilter` function to `genericFilterNaive`

25. Create and apply a truly generic function

```
var filterAirportsByIATA = (dest) -> airports filter ($.IATA == dest)
var genericFilterNaive = (a,f,v) -> a filter ($[f] == v)
var genericFilter = (a,fn) -> a filter fn($)
---
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
```

```
  airport: genericFilter(airports,(airport) -> airport.IATA == $.destination)
}
```

Those observant enough should be able to realize the `filter` function is generic enough and there is no need for yet another function. Nonetheless, this small exercise exposes us to aspects of functional programming and give us insights on how built-in functions that accept λ-expressions as arguments behave.
This is an example where a function is passed as an argument to yet another function. In a very similar manner we can return a function as the return value of a function–we shall see this next.

Optional: Curried functions

26. Create a curried function

```
var genericFilterC = (a) -> (fn) -> a filter fn($)
```

Nope there is no relationship to the spice, the name comes from Haskell Brooks Curry. Curry invented the concept of currying
There are two main reasons for creating curried functions:

- *Partial application of functions when all the arguments are not present at the same time–I have yet to find a use case where this partial application of curried functions can be applied in DW*
- *Function factories where out of a single definition you can define different but related functions. This last point is where curried functions can apply in DW.*

27. Define functions for filtering flights and airports out of the `genericFilterC` function

```
var genericFilterC = (a) -> (fn) -> a filter fn($)
var flightsFilter = genericFilterC(payload)
var airportsFilter = genericFilterC(airports)
```

Only one of the arguments is applied to `genericFilterC` and a function is returned and stored in the corresponding variables (i.e. `flightsFilt` and `airportsFilter`).
Another way to visualize these two function is by illustrating what they look like:

- *flightsFilter: `(fn) -> payload filter fn($)`*
- *airportsFilter: `(fn) -> airports filter fn($)`*

This discussion under stackoverflow could also help with your understanding.

28. Apply the `airportsFilter` function

```
map {
  ($),
  airport: airportsFilter((airport) -> airport.IATA == $.destination)
}
```

A more efficient transformation

29. Calculate the complexity of our algorithm
This is more of a discussion and a pen-and-paper exercise. The essence of this discussion is a calculation, the Big-O notation. Big-O notation is an abstract notation that measures the number of abstract operations that our algorithm will perform. Our calculation will be a function of the number of flights and the number of airports. Our focus is the iterations our algorithm performs, as such:

- *Assume we have `N` number of flights*
- *Assume we have `M` number of airports*
- *`2*M` operations from the expression that reads, parses, and obtaining the set of airports*
- *`8*N` operations from the expression that renames dynamically the fields*
- *`N*M` operations from the combination of flights and airports*

Thus the total number of operations is calculated by the `2*M + 8*N + N*M` polynomial. Remember the purpose of Big-O: abstractly identify the number of operations our algorithm performs to get the job done. Splitting your Big-O notation using polynomials such as the one we just created identifies areas of your algorithm that are candidates for optimization.
Replace `N` and `M` with a number of reasonable size (anything greater than 10 will do). Which one of the operations in the polynomial will result in the largest number? More importantly, if you keep calculating these operations with different number is the growth linear or exponential? If the growth is exponential then these portions of your algorithm are prime candidates for optimization. A linear growth is what you are after!
The growth of `N*M` is exponential or approaches an exponential growth. When performing lookups ensure we have atomic lookups as compared searching arrays. The ideal data structure to perform such atomic lookups is a map structure.

30. Go to the DW processor of the `mod4-read-parse-airports` flow
31. Change the airports array into an object (aka map)

```
%dw 2.0
output application/dw
---
readUrl(
  "classpath://airport-info.csv",
  "application/csv",
  {
    separator: ","
  }
)
distinctBy $.IATA
groupBy $.IATA
```

32. Explore the result in the preview
We can find the documentation page for `groupBy` here. In essence, `groupBy` takes the input array and transforms it into an object where the field names are set to the values the λ-expression `$.IATA` evaluates to.
33. Apply the `groupBy` function to the `airports` variable in `mod4-flights-airports` flow

```
var airports = readUrl(
  "classpath://airport-info.csv",
  "application/csv",
  {
    separator: ","
  }
)
distinctBy $.IATA
groupBy $.IATA
```

We should be getting an error now. The error identifies a type-mismatch on the `filter` function–i.e. An object is passed when an array is expected.

34. Change our expression to perform atomic lookups

```
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
  airport: airports[$.destination]
}
```

A couple of notes to conclude this WT:

- *The new complexity now is `3*M + 8*N + N`, we reached linear growth.*

- *The complexity we reached is related to processing time and we did not bother with the spatial complexity of our algorithm–i.e. the size of memory we use. Perhaps this will be a good exercise for the students to complete on their own.*
- *Finally, we wonder why go through creation of a generic filter function to begin with? The answer is we are getting exposure to functional programming where the knowledge thereof is crucial to all aspiring DW programmers.*

WT 4-3 Reordering fields

Create a new flow

1. Create a new flow named `mod4-reorder-object`
2. Drop a DW to the process area of the flow
3. Define the payload input metadata to `flight_json`
4. Edit the sample data
5. Turn on the preview
6. Change the output to `application/dw`

The `pluck` function

7. Use `pluck` to retrieve the values of an object

```
%dw 2.0
output application/dw
---
payload pluck (v,k,i) -> v
```

The documentation for `pluck` is found here. `pluck` is very similar to `mapObject`, we still iterate an object with it but instead of returning another object we are returning an array.

8. Modify the expression to retrieve the keys of an object

```
%dw 2.0
output application/dw
---
payload pluck (v,k,i) -> k
```

9. Modify the expression to retrieve the indexes

```
%dw 2.0
output application/dw
---
payload pluck (v,k,i) -> i
```

10. Repeat the last three steps but this time make use of the $, $$, and $$$ placeholders

```
%dw 2.0
output application/dw
---
payload pluck $
```

The manual just displays one of the expressions, we can form the other two on our own.
Can you notice the correlation between the number of $ and their position in the λ-expression? The same correlation is present in all function that support such placeholders.

The `reduce` function

11. Explore a `reduce` function

```
%dw 2.0
output application/dw
---
[3,1,2] reduce $$+$
```

12. Explore the result
    The result of this expression is just a summation of all numbers in the collection
    The documentation page for `reduce` does a pretty good job in explaining all the details.
    `reduce` falls under a family of functions generally referred to as fold functions. Unlike these fold functions which take any data as an input and result into any data as an output `reduce` in DW accepts only arrays as inputs but it can generate any type of data as an output.

13. Stop using the `$` and `$$` placeholders

```
%dw 2.0
output application/dw
---
[3,1,2] reduce (e, acc) -> acc + e
```

14. Provide an initial value for the accumulator

```
%dw 2.0
output application/dw
---
[3,1,2] reduce (e, acc=0) -> acc + e
```

This initial value on the accumulator is critical because now `reduce` knows what type of data to generate. We can replace this initial value to the empty array and inspect how the result changes from a number to the an array containing the exact same elements as the original array. This is because the `+` operator is overloaded to operated against numbers and arrays.

15. Trace the reduce function

- 1^{st} iteration: `(3,0) -> 0 + 3`
- 2^{nd} iteration: `(1,3) -> 3 + 1`
- 3^{rd} iteration: `(2,4) -> 2 + 4`

The secret in understanding `reduce` is understanding that the result of the previous iteration is used to set the accumulator for the next! The result of `reduce` is the result of the final iteration.
There are plenty of use-cases where `reduce` is ideal! You may as well just check the stackoverflow section for dataweave. Finally, we shall complete the reordering of an object using `reduce` because it will make for an optimal implementation vs when using `map`.

Reorder the fields

16. `reorder` function skeleton

```
%dw 2.0
output application/dw
var reorder = (o, ris) -> o
---
payload reorder [8,2,1,7,3,4,7,5,6,0]
```

The `reorder` function takes two arguments:

- *`o`: The object to reorder*

- *`ris`: An array of numbers containing the indexes of the new ordering for the object–that is the field in current position 8 will appear in position 0 of the re-ordered object, etc.*

17. Reorder using `map`

```
%dw 2.0
output application/dw
var reorder = (o, ris) -> do {
  var fs = o pluck $$
  ---
  {(ris map {
      (fs[$]): o[$]
    })}
}
---
payload reorder [8,2,1,7,3,4,7,5,6,0]
```

We made use of `map` to iterate the `ris` and the fact the objects can be accessed by indexes to perform the re-ordering. Finally, destroyed the array and collapsed the objects in it using the Dynamic elements feature.

18. Rename `reorder` to `reorderMap`

19. Reorder using `reduce`

```
var reorder = (o, ris) -> do {
  var fs = o pluck $$
  ---
  ris reduce (e,acc={}) -> acc ++ {
    (fs[e]): o[e]
  }
}
```

This solution is superior because there is no need to destroy arrays, collapse objects. We obtain the final result through a single iteration as compared to two from the previous solution.
Always consider what type of data we need to generate out of a collection–if it is anything but an array then `reduce` should be the way to go.

Integrate field re-ordering to our main use-case

20. Copy-n-paste the `reorder` function to `mod4-flights-airports`

```
var reorder = (o, ris) -> do {
  var fs = o pluck $$
  ---
  ris reduce (e,acc={}) -> acc ++ {
    (fs[e]): o[e]
  }
}
---
payload map (
$ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
  airport: airports[$.destination]
}
```

21. Re-order every single flight object in reverse

```
var reorder = (o, ris) -> do {
  var fs = o pluck $$
  ---
  ris reduce (e,acc={}) -> acc ++ {
    (fs[e]): o[e]
  }
}
---
payload map (
$ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
  airport: airports[$.destination]
}
map (
  $ reorder (8 to 0)
)
```

Module 5

Recursion

WT 5-1 Recursive summation

Create a new flow

1. Create a new Mule Configuration file and name it `mod5`
2. Create a new flow named `mod5-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. Recursive summation formula
   - Termination: $\Sigma(0) = 0$
   - Recursion: $\Sigma(n) = n + \Sigma(n-1)$
7. Define and apply the `recsum` function

```
%dw 2.0
output application/dw
var recsum = (n: Number) -> if (n <= 0) 0 else n + recsum(n - 1)
---
recsum(3)
```

The definition is very similar to the formula we laid out in the previous step. Such is the beauty of recursive functions, they closely reflect their textbook definitions.

8. `recsum` with any number over 254

```
%dw 2.0
output application/dw
var recsum = (n: Number) -> if (n <= 0) 0 else n + recsum(n - 1)
---
recsum(255)
```

9. The `stackoverflow` error

```
Stack Overflow. Max stack is 256
```

Those of us familiar with the stackoverflow errors we know what has just happened. When making recursive calls more memory is used because (in the Stack area of the memory) that stores the frame/environment details of your call. If we have enough recursive calls we run out of memory, i.e. stackoverflow errors.
In DW this limit is fixed to 256 depths preemptively to ensure a more graceful behavior. Depending on the type of recursive function we are writing we could increase this limit to a different number using a Mule Runtime startup option, for example `-M-Dcom.mulesoft.dw.stacksize=`
We can avoid such stackoverflow errors by writing tail-recursive functions.

Tail-recursion

10. Define and apply the `tailrecsum` function

```
%dw 2.0
output application/dw
var recsum = (n: Number) -> if (n <= 0) 0 else n + recsum(n - 1)
var tailrecsum = (n: Number, result: Number = 0) ->
                    if (n <= 0) result else tailrecsum(n - 1, result + n)
---
tailrecsum(2550)
```

A tail-recursive function is a function where the very last operation that takes place is the recursive call, hence its name. In order to build such a function the result of the function is usually piggybacked as an additional argument to the function. Compilers, including the DW compiler, can detect tail-recursive functions and optimize them such that they are no longer make use of stack memory!
In order case in order to simplify the invocation of our function we have set an initial value of `0` to this extra argument.
A word of warning, creating tail-recursive for recursive functions of sufficient complexity is considered hard. They require experience and would potentially make our code unreadable, unmaintainable. Create them when they are absolutely needed and stick to the more readable recursive version in all other cases.

11. Apply the `@TailRec()` annotation

```
@TailRec()
var tailrecsum = (n: Number, result: Number = 0) ->
                    if (n <= 0) result else tailrecsum(n - 1, result + n)
```

This annotation will confirm to us that we have defined a tail-recursive function. If you apply this annotation to a non-tail-recursive function the compiler will respond with an error.

WT 5-2 Recursive flatten

Create a new flow

1. Create a new flow named `mod5-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

```
%dw 2.0
output application/dw
---
[0,1,[2,[3,[4,[5]]]]]
```

6. Apply `flatten` four times

```
%dw 2.0
output application/dw
---
flatten(
  flatten(
    flatten(
      flatten(
        [0,1,[2,[3,[4,[5]]]]]
      )
    )
  )
)
```

The **rflatten** function

7. Create a recursive function that just traverses the arrays

```
%dw 2.0
output application/dw
var rflatten = (a: Array) -> a map (
  if (not ($ is Array)) $ else rflatten($)
)
---
rflatten([0,1,[2,[3,[4,[5]]]]])
```

To see that all elements in the arrays of sub-arrays are being traversed, we only need to change the second `$` in the if-expression to a literal value.

8. Apply `flatten` at the right level

```
%dw 2.0
output application/dw
var rflatten = (a: Array) -> flatten(a map (
  if (not ($ is Array)) $ else rflatten($)
))
---
rflatten([0,1,[2,[3,[4,[5]]]]])
```

A tail-recursive version of **rflatten**

9. Create a trail-recursive version of `rflatten`

```
@TailRec()
var tailrflatten = (a: Array) ->
  if (
    not (a dw::core::Arrays::some ($ is Array))
  ) a else tailrflatten(flatten(a))
---
tailrflatten([0,1,[2,[3,[4,[5]]]]])
```

This solution is ingenious because we don't even need to piggyback the result as an argument. Instead we recursively apply `flatten` for as long as the array contains arrays. This `tailrflatten` is a top-to-bottom solution as compared the bottom-up approach we take with the `rflatten` function.

WT 5-3 Traverse and transform the flights and airports data structure

Traverse the flights and airports data structure

1. Navigate to the `mod4-flights-airports` flow
2. Click on the DW processor in the source area of the flow
3. Define the `traverse` function handling arrays

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

Recursively, traverse each one of the elements in the array.

4. Overload `traverse` to handle objects

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

Recursively, traverse the each one of the keys and values of the object.

5. Overload `traverse` to handle keys

```
fun traverse(a: Array) = a map traverse($)
fun traverse(o: Object) = o mapObject { (traverse($$)): traverse($) }
fun traverse(k: Key) = k
```

For the time being just return the key as is. The reason is because we will apply the function and then we shall apply changes to these keys and visually identify our changes as we perform them.

6. Overload the `traverse` to handler strings

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

Again the time being just return the string as is for the same reason as above.
We only need to overload our `traverse` function for arrays, objects, keys, and strings because that is the only types we need to traverse. If we had additional types then we must provide additional overloaded function definitions.

7. Apply the `traverse` function

```
fun traverse(a: Array) = a map traverse($)
fun traverse(o: Object) = o mapObject { (traverse($$)): traverse($) }
fun traverse(k: Key) = k
fun traverse(s: String) = s
---
traverse(
  payload map (
    $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
  )
  map {
    ($),
    airport: airports[$.destination]
  }
  map (
    $ reorder (8 to 0)
  )
)
```

There should be no affect seen in the preview.

8. Remove the extra leading space to the `ICAO` field

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

Notice the how the space is removed for all airports. We are traversing the data!

9. Upper case all field names

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

We could potentially combine the field renaming we performed earlier on using this overloaded `traverse` function for keys. We have to be careful, however, because all fields independent of depth will be renamed.

10. Lower case all strings

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

11. Remove the `traverse` function application

```
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
  airport: airports[$.destination]
}
map (
  $ reorder (8 to 0)
)
```

A more flexible solution

12. Define the `traverseFn` operating over arrays

```
fun traverseFn(a: Array, fn) = a map traverseFn($,fn)
```

Very similar definition to `traverse` the only difference being an extra the argument `fn`. `This fn is a function that will contain the operations to be performed for the simple types, i.e. keys and strings. This way we decouple operations from simple types and the complex types which we primarily need in order to traverse.`

13. Overload the `traverseFn` operating over objects

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

14. Overload the `traverseFn` operating over keys

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

Because we decouple the simple type from the complex type expressions and provide these simple type expressions through a λ-expression we only need to apply this λ-expression for keys.

15. Overload the `traverseFn` operating over strings

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

16. Apply `traverseFn`

```
fun traverseFn(a: Array, fn) = a map traverseFn($,fn)
fun traverseFn(o: Object, fn) = o mapObject { (traverseFn($$,fn)): traverseFn($,fn) }
fun traverseFn(k: Key, fn) = fn(k)
fun traverseFn(s: String, fn) = fn(s)
---
payload map (
  $ mapObject (v,k,i) -> {(fs2rn[k] default k): v}
)
map {
  ($),
  airport: airports[$.destination]
}
map (
  $ reorder (8 to 0)
)
traverseFn (
  (e) -> e match {
    else -> $
  }
)
```

This λ-expression is just a `match` operator making no changes currently to the keys and strings. We will make this changes next.

17. Fix the `ICAO` and upper case all fields again

```
traverseFn (
  (e) -> e match {
    case k if (k is Key) -> upper(trim(k))
    else -> $
  }
)
```

18. Lower case all strings again

```
traverseFn (
  (e) -> e match {
    case k if (k is Key) -> upper(trim(k))
    case s if (s is String) -> lower(s)
    else -> $
  }
)
```

Cast to **`Number`** and **`Date`** when possible

19. Explore the output data in the preview
    There are plenty of values that are numbers and could be typecasted into a number as compared to just the string representation of that number. Similarly, there is the `DATE` field that contains a date in a string denotation.

20. Apply `dw::Runtime::try` and try to typecast numbers

```
traverseFn (
  (e) -> e match {
    case k if (k is Key) -> upper(trim(k))
    case s if (s is String) -> dw::Runtime::try(() -> s as Number)
    else -> $
  }
)
```

Every single value in our objects is now a `TryResult` structure. Those with `success: true` contain the type-casted number, while those with `success: false` contain the error details.

21. Chain the `dw::Runtime::orElse` function

```
traverseFn (
  (e) -> e match {
    case k if (k is Key) -> upper(trim(k))
    case s if (s is String) -> dw::Runtime::try(() -> s as Number)
                               dw::Runtime::orElse () -> lower(s)
    else -> $
  }
)
```

The documentation for `dw::Runtime::orElse` can be found here.

22. Apply `dw::Runtime::orElseTry` to typecast the `DATE` field

```
traverseFn (
  (e) -> e match {
    case k if (k is Key) -> upper(trim(k))
    case s if (s is String) ->
        dw::Runtime::try(() -> s as Number)
        dw::Runtime::orElseTry (() -> s as Date {format: "yyyy/MM/dd"})
        dw::Runtime::orElse () -> lower(s)
    else -> $
  }
)
```

The documentation for `dw::Runtime::orElseTry` can be found here