Module 5: Recursion

Goal

Inputs

```
[
  {
    "airlineName": "Delta",
    "code": "A1B2C3",
    "departureDate": "2018/03/20",
    "destination": "SFO",
    "emptySeats": "40",
    "origin": "MUA",
    "planeType": "Boing 737",
    "price": "400.0"
  },
  …
]
```

```
openFlightsAirportId,airportName,city,country,IATA
492,London Luton Airport,London,UK,LTN
499,Jersey Airport,Jersey,Jersey,JER502,London
Gatwick Airport,London,UK,LGW
…
```

Expression

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

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 {
    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 -> $
  }
)
```

Output

```
[
 {
  AIRPORT: [
   {
    OPENFLIGHTSAIRPORTID: 3469,
    AIRPORTNAME: "san francisco international airport",
    CITY: "san francisco",
    COUNTRY: "united states",
    IATA: "sfo",
    ICAO: "ksfo",
    LONGITUDE: 37.61899948,
    LATITUDE: -122.375,
    ALTITUDE: 13,
    TIMEZONE: -8,
    DST: "a",
    TIMEZONE: "america/los_angeles",
    TYPE: "airport",
    SOURCE: "ourairports"
   }
  ],
  PRICE: 400.0,
  PLANE: "boing 737",
  ORIGIN: "mua",
  SEATS: 40,
  DESTINATION: "sfo",
  DATE: |2018-03-20| as Date {format: "yyyy/MM/dd"},
  CODE: "a1b2c3",
  CARRIER: "delta"
 },
 ...
]
```

At the end of this module, you should be able to

- Write recursive functions
- StackOverflow errors and how to overcome them
- Understand tail-recursion
- Recursively flatten arrays containing sub-arrays
- Traverse any input data-structure

Recursive functions

Recursion and its drawbacks

- Recursion is used when a problem is defined in terms of itself
  - Tree data structures
  - Summation: S(n) = N + S(n-1)
  - Fibonacci numbers: F(n) = F(n-1) + F(n-2)
  - Recursion is very similar to induction, for those mathematically inclined
    - That is there is a base case, we call it the exit or terminating condition in recursion
    - Inductive step, we call it recursive step in recursion
- StackOverflow errors
  - Very common in recursion
  - Every recursive call uses Stack memory to store its environment
    - Environment is the set of values of all variables, arguments, etcwhen you make a function call
  - If recursion does not terminate or does not terminate quickly enough we will run out of stak memory—hence the StackOverflow errors.

Overcoming stackoverflow errors

- Brute-force
  - In DataWeave the default StackOverflow error is thrown after 256 recursions
  - Increase this limit with the `com.mulesoft.dw.stacksize` startup property
  - In studio `-M-Dcom.mulesoft.dw.stacksize=1000`
  - Ensure you have enough memory for such an increase
- Tail recursion/call
  - This is language optimization
  - It is enabled if the very last operation in the function is the recursive call
  - Often enough such a modification requires to piggy-back the result as an extra argument in the function
  - Creating a tail recursive function is non-trivial for problems with enough complexity
  - Annotate tail recursive functions in DW with the `@TailRec()` annotation
    - DW will throw compilation errors if your function is not tail recursive

Walkthrough 5-1: Recursive summation

- Create a recursive summation function where
  - The exit condition is S(0) = 0
  - The recursive call is S(n) = n + S(n – 1)
- Illustrate the StackOverflow errors
- Create a tail recursive function of the summation function
- Use the `@TailRec()` annotation

Recursive flatten

Recursive flatten

- `flatten(Array): Array`
  - Removes the first level of inner sub-arrays at a time
  - [0,1,[2,[3,[4,[5,]]]]] needs 4 applications of flatten
- `rflatten(Array): Array`
  - Base case: is the termination of iterating an array
  - Recursive step: invoke `rflatten` if the element in the array is itself an array
- `tailrflatten(Array): Array`
  - Base case: no elements in the array are arrays, thus returning the array
  - Recursive step: `flatten` one level of the sub-arrays and call `tailrflatten` on the result

Walkthrough 5-2: Recursive flatten

- Create the `rflatten` function
- Create the `tailrflatten` function

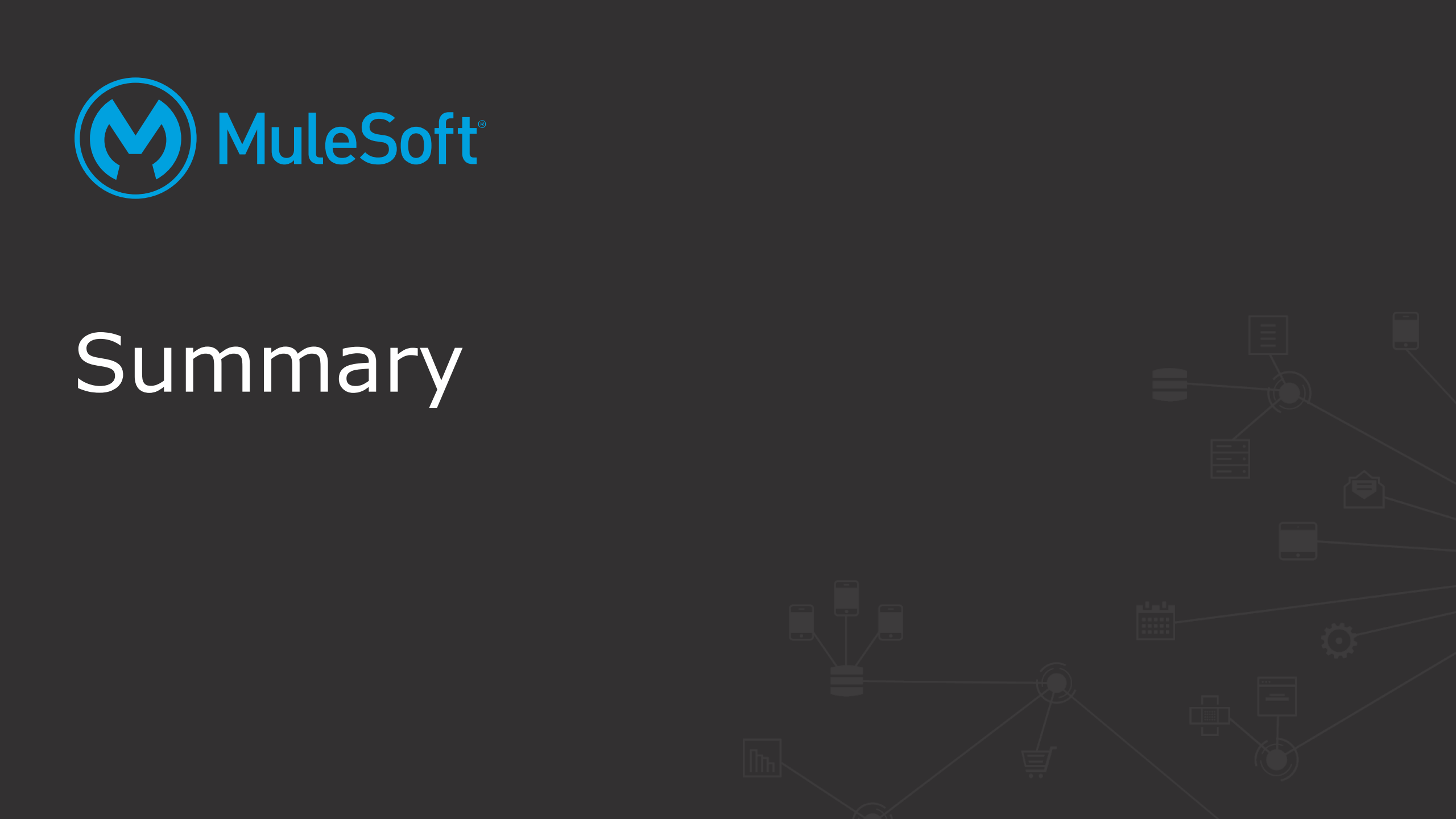
Traversing arbitrary data structures

Traversal functions

- Create a set of overloaded functions
  - One for each type of data you expect
  - Iterate over arrays and traverse each one of the elements
  - Iterate over objects and traverse each one of the keys and the values
  - Transform the simple data, such as numbers, strings, keys, etc.
- A more flexible solution
  - Maintains the iteration of complex types such as arrays and objects
  - Decouples simple type operations using a lambda-expression

Walkthrough 5-3: Traverse the flights and airports

- Create and apply a traverse function specific for the flights and airports data structure
- Optional: Create a more general traverse function where
  - the simple types, such as keys and strings are transformed using a lambda-expression
  - This lambda-expression is passed as an argument to the traverse function



Summary

Summary

- Recursion is used when a problem is defined in terms of itself
- Recursive functions are defined in terms of the (1) terminating condition and (2) the recursive step
- StackOverflow errors are thrown when stack memory is exhausted or when the DW limit is met
  - The default limit is set to 256
- Increase the limit through the `com.mulesoft.dw.stacksize` startup property
- Define tail recursive or tail call functions that DW optimizes
- By putting everything we learned in this class together we are able to traverse any data structure—if we are able to traverse, we are able to transform