

Module 3: Defensive Programming

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



- Overload functions
- Apply pattern matching
- Catch and process errors
- Retrieve partial results

What is Defensive Programming



- Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances.¹
- Defensive programming results in
 - Less software bugs
 - Code is more maintainable
 - Even in the presence of bad data software will behave predictably

1. https://en.wikipedia.org/wiki/Defensive_programming



Function Oveloading

DataWeave's Typing System



- Polymorphism
 - A single declaration can deal with multiple types as inputs
 - Polymorphic classes
 - Ad hoc polymorphism

```
- fun id(e) = e
```

Parametric polymorphism (out of scope)

```
- fun idT>(e: T) = e
```

- Subtyping (out of scope)
 - Applies to OOP, i.e. inheritance
- The Null type
 - null is Null
 - Such typing system allows for catching "NPE"s at compile time!

Function overloading



- Can only be done using the fun syntax
 - i.e. no overloading for lambda-expressions
- Keep declaring functions with
 - the same name
 - the arguments along with the argument types
 - different types on the arguments

For example:

```
fun id(n: Null) = n
fun id(a: Array) = a
```

Walkthrough 3-1: Function Overloading



- Create a function to match the Null input
- Overload the function to accept arrays
- Overload the function to accept objects
- Overload the function to accept strings
- Overload the function to accept numbers
- Inspect the signature of functions in the preview



Pattern Matching

Pattern Matching



- Match literal values or patterns of data and perform the corresponding action
- Literal Pattern Matching

```
10 match {
  case 10 -> "Ten"
  else -> "No match"
}
```

Type Pattern Matching

```
10 match {
  case is Number -> "Number Found"
  else -> {message: "No match", value: $}
}
```

Any conditional Pattern Matching

```
10 match {
  case n if (is Number and n >= 10 ) -> "Number Found greater than 10"
  else v -> {message: "No match", value: v}
}
```

9

Walkthrough 2-2: Pattern Matching



- Apply pattern matching to literal values
- Apply pattern matching using type checking
- Apply pattern matching using any conditionals
- Apply user defined placeholders for every case
- Retrieve a default value when no match is found



Error Handling

Error Handling



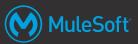
- dw::Runtime::try(() -> T):TryResult<T>
 - Accepts a lambda-expression that takes no arguments and contains the expression to test for correctness
 - This is a delegate function, because it delegates the execution of the expression at a different level
 - In functional programming only through functions you pass code to be executed at a different level
 - Returns a TryResult data structure
 - Contains a success field that when
 - Set to true you will have another field named result containing the result of the expression
 - Set to false you will have an error data structure containing meta-data about the error.

Walkthrough 3-3: Error Handling



- Explore the documentation of the dw::Runtime::try() function
- Catch and handle division-by-zero errors
- Chain pattern matching to fine tune the error-handling code
- Refactor the error-handling code in a function
- Apply the function to an expression

Walkthrough 3-4: Optional — Partial Results



- Process collections of data
- Accommodate for bad data
- Retrieve partial results for the good data
- Enhance the results to also contain the bad data



Summary



Summary



- Defensive programming is the a design discipline use to allow for maintainable software with less bugs to behave predictably even in the presence of bad data.
- Defensive programming can be exercised in DataWeave through
 - The typing system
 - Function overloading
 - Pattern matching
 - Error handling