WWDC 19
World Wide Developer Conference

Core Language Enhancements

~Delane Freeman
email:delane.freeman@4js.com

Banyan Tree Spa Mayakoba
30 September - 3 October 2019

FOUR Js
The Power of Simplicity

What's this presentation about?

The focus of this presentation

- **Modernizing your code with Genero 3.20**
  - Genero Core Language Enhancements
    - Passing RECORD variables by reference to functions
    - Variable Initialization
    - Named parameters in function calls
    - Record TYPE Methods
    - INTERFACE - Achieving polymorphism
    - Miscellaneous
  - Coding design/choices
    - Pictorials
    - Code snippets
    - A demonstration

What's this presentation about?

What will be left out

- **Subjects not covered**
  - Genero RESTful Framework
  - Changes for Genero Mobile

Passing RECORD variables by reference

Synopsis

- **Previously: passing by reference only possible for**
  - DYNAMIC ARRAY, DICTIONARY
  - BYTE, TEXT and objects such as base.Channel or om.DomNode
- **New: RECORD variable types can be passed by reference**
- **New: keyword "INOUT"**
  - Must be specified in the function call parameter
  - FUNCTION functionName(recordName dataType INOUT)
  - Must be called
    - with: CALL functionName(recordName)
    - not: CALL functionName(recordName.*)

Passing RECORD variables by reference(cont.)

Sample

```
 9
10  TYPE cust_type RECORD
11      pkey INTEGER,
12      nm VARCHAR(50),
13      addr VARCHAR(100),
14      crea DATE
15  END RECORD
16
17  MAIN
18
19      DEFINE rec cust_type
20
21      LET rec.pkey = 834
22      LET rec.nm = "Mike Torn"
23      LET rec.crea = MDY(12, 24, 2018)
24
25      -- Legacy call passing all record members on the stack
26      CALL func_rec_by_val(rec.*) -- .* expands all record members
27      DISPLAY "1:", rec.*
28
29      -- Passing record as reference allows to modify it
30      CALL func_rec_by_ref(rec) -- Note that .* is not used here!
31      DISPLAY "2:", rec.*
32
33  END MAIN
34
```

Passing RECORD variables by reference(cont.)

Sample

```
34
35  FUNCTION func_rec_by_val(r cust_type)
36      INITIALIZE r.* TO NULL
37      LET r.pkey = 999
38      LET r.nm = "<undefined>"
39  END FUNCTION
40
41  FUNCTION func_rec_by_ref(r cust_type INOUT)
42      INITIALIZE r.* TO NULL
43      LET r.pkey = 999
44      LET r.nm = "<undefined>"
45  END FUNCTION
46
47
48
49
50
51
52
53
54
55
56
57
58
59
```

Passing RECORD variables by reference(cont.)

Caution

- **RECORD types *must* match**
  - Native types
  - Extended types STRING, util.JSON, etc.
  - User TYPEs
    - TYPE A record vs. TYPE B record definitions
    - It doesn't matter that both type definitions contain the same members
    - TYPE A is “a” and TYPE B is “b”; therefore, a!=b
- **The compiler must be able to resolve the reference definition**
  - The FUNCTION must be in the same module; or,
  - The FUNCTION must be in a module imported by IMPORT FGL

Variable Initialization

Synopsis

- **Previously: Many "LET" assignment statements or util.JSON.parse() to initialize variables...**
- **New: Definition initialization syntax**
  - Initializer for scalar variables
  - Initializer for RECORD (with auto completion for RECORD members)
  - Initializer for ARRAY
  - Code completion for RECORD members , KEYWORDS

Variable Initialization(cont.)

Sample (Simple Type)

```
9
10  MAIN
11      DEFINE s1 STRING = "This is a string"
12      DEFINE i1 INTEGER = -999
13      DEFINE d1 DATE = MDY(12, 24, 2018)
14
15      DISPLAY s1
16      DISPLAY i1
17      DISPLAY d1
18
19  END MAIN
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
```

Variable Initialization(cont.)

Sample (Record)

```
 9
10  TYPE type1 RECORD
11      pkey INTEGER,
12      nm VARCHAR(50),
13      addr VARCHAR(100),
14      crea DATE,
15      orders DYNAMIC ARRAY OF INTEGER
16  END RECORD
17
18  MAIN
19      DEFINE rec1
20          type1 -- All members are initialized
21          = (pkey: 834,
22             nm: "Mike Torn",
23             addr: "5 Big Mountain St.",
24             crea: MDY(12, 24, 1997),
25             orders:[234, 435, 456])
26      DEFINE rec2
27          type1 -- Some members are initialized
28          = (pkey: 0, nm: "<undefined>", addr: "<undefined>")
29
30      DISPLAY rec1.nm, rec1.crea
31      DISPLAY rec1.orders.getLength()
32
33      DISPLAY rec2.pkey, rec2.nm
34
```

Variable Initialization(cont.)

Sample (Dynamic Array)

```
9
10  MAIN
11      DEFINE arr1 DYNAMIC ARRAY OF RECORD
12          pkey INTEGER,
13          name VARCHAR(50)
14      END RECORD
15          = [(pkey: 834, name: "Mike Torn"),
16             (pkey: 981, name: "Blake Crystal"),
17             (pkey: 993, name: "Tom Yorp")]
18
19      DISPLAY arr1.getLength()
20
21  END MAIN
22
23
24
25
26
27
28
29
30
31
32
33
```

Variable Initialization(cont.)

Caution

- **Must maintain type compatibility**
  - Compiler will throw the error -6631, if the type of the initializer and the type of the variable are incompatible:
  - Initializers can't use expressions like (a+b)
    - Globals and module global variables are initialized before MAIN: there is no concept of error handling
    - Errors can not be caught by TRY/CATCH: DEFINE is not a statement

Named parameter in function calls

Synopsis

- Allows labeling the parameters of function calls with the name of the parameter which was used in the declaration of the function.
- Advantage: much better readability of the source code, automatic code completion
- Function parameter names are optional

Named parameter in function calls(cont.)

Caution

- **Names must match exactly with the function definition**
- **Order must be maintained as in the definition**
- **Cannot omit parameters(there are no default arguments)**
- **The compiler must be able to resolve the reference definition**
  - The FUNCTION must be in the same module; or,
  - The FUNCTION must be in a module imported by IMPORT FGL

Named parameter in function calls(cont.)

Sample

```
 9
10  MAIN
11      CALL func1(id: 999, description: "This is a demo", rate: 1.34)
12
13      --> name does not match
14      CALL func1(id:999, desc: "This is a demo", rate: 1.34)
15      --> order does not match
16      CALL func1(description: "This is a demo", rate: 1.34, id:999)
17      --> omitted "rate"
18      CALL func1(id:999, description: "This is a demo")
19
20  END MAIN
21
22  FUNCTION func1(id INTEGER, description STRING, rate FLOAT)
23      DEFINE sb base.StringBuffer
24
25      --> Can be used with built-in libraries
26      LET sb = base.StringBuffer.create()
27      CALL sb.append(str:description)
28      CALL sb.replace(oldStr:"foo", newStr:"bar", occurrences:0)
29
30      DISPLAY id, description, rate
31  END function
32
33
34
```

TYPE Methods

Synopsis

- **Methods are functions that perform on a variable with a specific user-defined type**
- **Use methods to implement the interface (the access methods) for a type**
- **Allows you to write robust code like in Object-Oriented Programming languages, without the complexity and traps of OOP**
- **A Method is:**
  - A function name
  - Defined with a receiver argument specified in parentheses before the function name
    - consists of an identifier followed by a user-defined type
    - is then referenced in the function body as the target

TYPE Methods(cont.)

Sample

```
10  PUBLIC TYPE cust_type RECORD
11      pkey INTEGER,
12      nm VARCHAR(50),
13      addr VARCHAR(100),
14      crea DATETIME YEAR TO FRACTION(5),
15      modi DATETIME YEAR TO FRACTION(5),
16      orders DYNAMIC ARRAY OF INTEGER
17  END RECORD
18
19  PUBLIC FUNCTION (r cust_type) initializeWithDefaults(name STRING) RETURNS ()
20      INITIALIZE r.* TO NULL
21      LET r.pkey = 999
22      LET r.nm = name
23      LET r.addr = "<undefined>"
24      LET r.crea = CURRENT
25  END FUNCTION
26
27  PUBLIC FUNCTION (r cust_type) setAddress(addr STRING) RETURNS ()
28      LET r.modi = CURRENT
29      LET r.addr = addr
30  END FUNCTION
31
32  PUBLIC FUNCTION (r cust_type) addOrder(ordid INTEGER) RETURNS ()
33      LET r.modi = CURRENT
34      CALL r.orders.appendElement()
35      LET r.orders[r.orders.getLength()] = ordid
```

TYPE Methods(cont.)

Sample

```
41
42  FUNCTION main()
43
44      DEFINE c1 cust_type
45
46      --> Call to initializer method
47      CALL c1.initializeWithDefaults("Mike Torn")
48      DISPLAY c1.pkey, c1.nm
49      --> Call to setter method
50      CALL c1.setAddress("5 Matchita St.")
51      DISPLAY c1.modi, c1.addr
52      --> Calls to mutator methods
53      CALL c1.addOrder(485)
54      CALL c1.addOrder(948)
55      DISPLAY c1.modi, c1.getOrderCount()
56
57  END FUNCTION
58
59
60
61
62
63
64
65
66
```

TYPE Methods(cont.)

Caution

- The receiver-type must be a TYPE defined in the same module as the method
- The receiver-type must define a RECORD type.
- Method names and field names must be distinct (It is not legal to define a method with the same name as a field of the receiver-type).
- A method must be called by referencing a variable type of receiver-type followed by dot method-name and parameters
- No designated constructor, just write an *init* function

INTERFACEs - Achieving Polymorphism

What is "polymorphism"?

- **Noun: the condition of occurring in several different forms**
- **Occurs when there is a hierarchy related by inheritance**
- **Allows routines to use variables of different TYPEs at different times**
- **In Genero a call to a member function will cause a different function to be executed depending on the TYPE of the object that invokes the function**
- **Implemented through INTERFACE and TYPE Methods**

INTERFACEs - Achieving Polymorphism

Synopsis

- INTERFACEs allow code to specify that behavior(s) is(are) required
- Behavior is defined by a set of METHODs
- No particular implementation is enforced. It's sufficient that the INTERFACE defines methods with required names and signatures
- Values of a TYPE can be passed to any function accepting an INTERFACE as parameter
- TYPEs can implement multiple INTERFACEs
- The same INTERFACE can be implemented by many TYPEs (polymorphism)
- ARRAY and DICTIONARY accept interface values

INTERFACEs - Achieving Polymorphism(cont.)

Sample -INTERFACEs allow code to specify that behavior(s) is(are) required

```
11
12  TYPE Rectangle RECORD
13      height, width FLOAT
14  END RECORD
15
16  TYPE Circle RECORD
17      diameter FLOAT
18  END RECORD
19
20  TYPE Shape INTERFACE
21      area() RETURNS FLOAT,
22      kind() RETURNS STRING
23  END INTERFACE
24
25  FUNCTION (r Rectangle) area() RETURNS FLOAT
26      RETURN r.height * r.width
27  END FUNCTION
28
29  FUNCTION (r Rectangle) kind() RETURNS STRING
30      RETURN "Rectangle"
31  END FUNCTION
32
33  FUNCTION (c Circle) area() RETURNS FLOAT
34      RETURN util.Math.pi() * (c.diameter / 2) ** 2
35  END FUNCTION
36
```

INTERFACEs - Achieving Polymorphism(cont.)

Sample - Behavior is defined by a set of METHODs

```
19
20  TYPE Shape INTERFACE
21      area() RETURNS FLOAT,
22      kind() RETURNS STRING
23  END INTERFACE
24
25  FUNCTION (r Rectangle) area() RETURNS FLOAT
26      RETURN r.height * r.width
27  END FUNCTION
28
29  FUNCTION (r Rectangle) kind() RETURNS STRING
30      RETURN "Rectangle"
31  END FUNCTION
32
33  FUNCTION (c Circle) area() RETURNS FLOAT
34      RETURN util.Math.pi() * (c.diameter / 2) ** 2
35  END FUNCTION
36
37  FUNCTION (r Circle) kind() RETURNS STRING
38      RETURN "Circle"
39  END FUNCTION
40
41
42
43
44
```

INTERFACEs - Achieving Polymorphism(cont.)

Sample - The same INTERFACE can be implemented by many TYPEs

```
48
49  FUNCTION totalArea(shapes DYNAMIC ARRAY OF Shape) RETURNS FLOAT
50      DEFINE i INT
51      DEFINE area FLOAT
52      FOR i = 1 TO shapes.getLength()
53          LET area = area + shapes[i].area()
54      END FOR
55      RETURN area
56  END FUNCTION
57
58  FUNCTION main()
59      DEFINE r1 Rectangle = (height: 10, width: 20)
60      DEFINE c1 Circle = (diameter: 10)
61      DEFINE shapes DYNAMIC ARRAY OF Shape
62
63      LET shapes[1] = r1
64      LET shapes[2] = c1
65
66      DISPLAY shapes[1].kind(), shapes[1].area()
67      DISPLAY shapes[2].kind(), shapes[2].area()
68      DISPLAY "Total:", totalArea(shapes)
69
70  END FUNCTION
71
72
73
```

IMPORT FGL

Synopsis

- **Circular dependency with IMPORT FGL**
  - Occur when several modules reference each other with IMPORT FGL
  - Previously produced the compilation error -8402
- **New tool to help migrate applications linked traditionally "fglrun *--print-imports*"**
  - Attempts to resolve all symbols as done during linking
  - Lists the IMPORT FGL instructions required in each module to avoid linking requirement
- **Required for certain new features**
  - Passing RECORD variables by reference
  - Named parameter in function calls

Miscellaneous

Source code formatting

- **Source code formatting with *fglcomp --format***

  --fo-inplace: Write formatted output back to the provided file, instead of stdout. Creates a copy of the original file in filename.4gl~

  --fo-column-limit=integer: Define the source line width limit. Default is 80

  --fo-indent-width=integer: Number of columns to use for indentation. Default is 4

  --fo-continuation-indent-width=integer: Indent width for line continuations. Default is 4.

Miscellaneous

Source code formatting (cont.)

--fo-label-indent={0|1}: When 1, indent instruction clauses such as WHEN in a CASE instruction. Default is 1 (enabled)

--fo-pack={0|1}: When 1, try to put as much items on the same line. When 0, use one line for each item. Default is 0 (do not pack)

--fo-lowercase-keywords={0|1}: When 1, produce lowercase keywords. When 0, produce uppercase keywords. Default is 0 (uppercase).

Miscellaneous

Source code formatting (cont.)

- **All --fo-* formatting options except --fo-inplace can be specified in a configuration file named .fgl-format.**
- **This configuration file is typically placed in the top directory of your project, to apply the same formatting rules to all sources located under this root directory**
- **SAMPLE .fgl-format**

```
1
2   # My code formatting options
3   --fo-column-limit=80
4   --fo-indent-width=4
5   --fo-continuation-indent-width=4
6   --fo-label-indent=1
```

Miscellaneous

Database support

- **Support for Oracle release 18c (private temp tables, driver: dbmora_18).**
- **Miscellaneous enhancements**
  - Constructs support [a-z]* lists/ranges
  - PostgreSQL: Specifying the schema in fglprofile
  - FGLPROFILE entry to avoid last serial retrieval when not needed
  - New utility function dbutl.db_get_last_serial().
- **SAP HANA 2.0 support**
- **Informix IDS 14.10 / CSDK 4.50**
- **Support for Informix trusted sessions**

Miscellaneous

Miscellaneous

- **trimWhiteSpace()**
  - Implemented for String and StringBuffer objects
    - trimWhiteSpace() considers whitespace as all characters less than or equal to blank space (ASCII(32))
    - This includes tab (\t), newline (\n), carriage-return (\r) and form-feed (\f)
    - As a best practice, consider replacing trim() with trimWhiteSpace() in your sources
- **Function definition supports empty RETURNS()**
  - RETURNS () allows standardization of code where every FUNCTION has a RETURNS statement
  - discovers errors at compile time and prevents runtime failures
- **Array/Record assignment supported with ".*" notation**
- **Compile multiple source files with singular *fglcomp* command**

Miscellaneous

Miscellaneous (cont.)

- **Enhanced client resource caching**
  - Previously: client side caching default:off
    - Could only be enable with GDC admin mode
    - May get unwanted cached resources
  - New: client side caching default:on
    - only if file size and modification time of a VM side resource change (image/font) the resource is re-transmitted (similar to rcp -p)
    - Beneficial for UR mode:
      - GBC needs to be transmitted only once
      - Only modifications/customizations require a re-transmit.

Miscellaneous

Miscellaneous (cont.)

- **VIM plugin options**
  - Disable case sensitivity for keywords
    - let fgl_ignore_case=1
  - Enable lowercase keywords in code completion proposals
    - let fgl_lowercase_keywords=1
    - Implies implicitly fgl_ignore_case=1
  - Format/beautify the source code on file save

Putting it all together...

Re-thinking the Officestore RESTful(low-level) web service

Source: https://github.com/FourjsGenero/ex_wwdc19_language_core

Time for a demo

Time for a demo

Demo - Implementing INTERFACE

Before - Multiple conditional checks and marshaling tasks

```
100         # Determine the method(verb) and send request to the factory
101         CASE incomingRequest.getMethod()
102         # The general convention is that GET is a query and performs no update.
103         WHEN "GET"
104             CALL factoryRestInterface.marshalRestGet(incomingRequest)
105
106         # POST generally is reserved for create operation
107         WHEN "POST"
108             CALL factoryRestInterface.marshalRestPost(incomingRequest)
109
110         # PUT generally is reserved for update operations
111         WHEN "PUT"
112             CALL factoryRestInterface.marshalRestPut(incomingRequest)
113
114         # DELETE generally is reserved for delete operations
115         WHEN "DELETE"
116             CALL factoryRestInterface.marshalRestDelete(incomingRequest)
117
118         OTHERWISE
119             LET applicationError = SFMT("<MAIN>Method not allowed: [%1]", incomi
120             CALL logger.logEvent(logger._LOGMSG,ARG_VAL(0),SFMT("Line: %1",__LIN
121             CALL incomingRequest.setResponseHeader("Content-Type","application/j
122             CALL incomingRequest.setResponseHeader("Cache","no-cache")
123             #LET errorMessage.description = applicationError
124             CALL incomingRequest.sendTextResponse(HTTP_NOTALLOWED,incomingReques
125         END CASE
```

Demo - Implementing INTERFACE

Before - Multiple conditionals and function calls

```
165         # Process account list query; assume query appears after the "?";
166             # i.e. for "accounts?user=bloggs" would be "bloggs"
167             WHEN ( "accounts" )
168                 CALL restAccountFactory.processQuery(queryFilter)
169                 RETURNING statusCode, factoryResponse
170
171             # Process category list query; assume query appears after the "?";
172             # i.e. for "categories?catnum=SUPPLIES" would be "SUPPLIES"
173             WHEN ( "categories" )
174                 CALL restCategoryFactory.processQuery(queryFilter)
175                 RETURNING statusCode, factoryResponse
176
177             # Process orderItem list query; assume query appears after the "?";
178             # i.e. for "items?itemnum=AR-001-A" would be "AR-001-A"
179             WHEN ( "items" )
180                 CALL restItemFactory.processQuery(queryFilter)
181                 RETURNING statusCode, factoryResponse
182
183             # Process order list query; assume query appears after the "?";
184             # i.e. for "orders?ordernum=952121" would be "952121"
185             WHEN ( "orders" )
186                 CALL restOrderFactory.processQuery(queryFilter)
187                 RETURNING statusCode, factoryResponse
188
189             # Process item list query; assume query appears after the "?";
190             # i.e. for "orderItem?ordernum=7" would be "7"
```

Demo - Implementing INTERFACE

After - Implement a DICTIONARY of INTERFACEs

```
39
40  # Service INTERFACEs
41  import fgl categoryInterface
42  import fgl supplierInterface
43  import fgl countryInterface
44
45  # Resource factory interface(basic C.R.U.D. functionality)
46  type resourceInterface interface
47      createResource(requestPayload string) returns(),
48      retrieveResource(requestPayload string) returns(),
49      updateResource(requestPayload string) returns(),
50      deleteResource(requestPayload string) returns()
51  end interface
52
53  # Dictionary of resource responses
54  define interfaceResponse dictionary of resourceInterface
55
56  # Response interface initializer
57  function initResponseInterface()
58      let interfaceResponse["categories"] = categoryInterfaceObject
59      let interfaceResponse["suppliers"] = supplierInterfaceObject
60      let interfaceResponse["countries"] = countryInterfaceObject
61  end function
```

Demo - Implementing INTERFACE

After - Defining the objects(TYPEs and Method functions) used in the INTERFACE

```
15
16  import util
17
18  import fgl http
19  import fgl logger
20  import fgl interfaceRequest
21
22  import fgl country
23
24  schema officestore
25
26  public type countryType record
27      code like country.code,
28      codedesc like country.codedesc
29  end record
30
31  public define countryInterfaceObject countryType
32
33  public function (this countryType) retrieveResource(requestPayload string) returns
34      define thisJSONArr util.JSONArray
35      define i, queryException integer
36
37      define query dynamic array of record
```

Demo - Implementing INTERFACE

After - Bringing it all together and calling by “reference”

```
91
92      # Check if resource is valid
93      if (isValidResource(requestResource))
94      then
95
96          # Get the request payload(data/query)
97          let requestPayload = util.JSON.stringify(getRequestItems())
98
99          # Process the request by named resource
100         case requestMethod
101             when C_HTTP_GET
102                 call interfaceResponse[requestResource].getResource(requestPaylo
103
104             when C_HTTP_POST
105                 call interfaceResponse[requestResource].createResource(requestPa
106
107             when C_HTTP_PUT
108                 call interfaceResponse[requestResource].updateResource(requestPa
109
110             when C_HTTP_DELETE
111                 call interfaceResponse[requestResource].deleteResource(requestPa
112
113             otherwise
114                 # Method not allowed with web service
115                 let applicationError = sfmt("Method(%1) not allowed.", requestMe
116                 call interfaceRequest.setResponse(C_HTTP_NOTALLOWED, "ERROR", ap
117                 call logger.logEvent(logger.C_LOGMSG, ARG_VAL(0), sfmt("Line: %1
118         end case
119
```

Upgrading with Genero 3.20

Some parting thoughts...

- New features appeal to younger developers
- Libraries can be enhanced with TYPE methods
- To get most of new features, consider IMPORT FGL and not linking(required for certain features)
- Code can be tidied and facilitate patch and merge if using a standard format(RETURNs, code beautifier)
- Function definitions using RETURNS can capture code errors at compile time

Core Language Enhancements

Presented by: Delane Freeman – FourJs Development Inc

Q & A

Core Language Enhancements

Thank you…

~Delane Freeman
email:delane.freeman@4js.com

Banyan Tree Spa Mayakoba
30 September - 3 October 2019

FOUR Js
The Power of Simplicity