Take Advantage of RPG's

DATA-GEN

Presented by

Scott Klement

http://www.scottklement.com

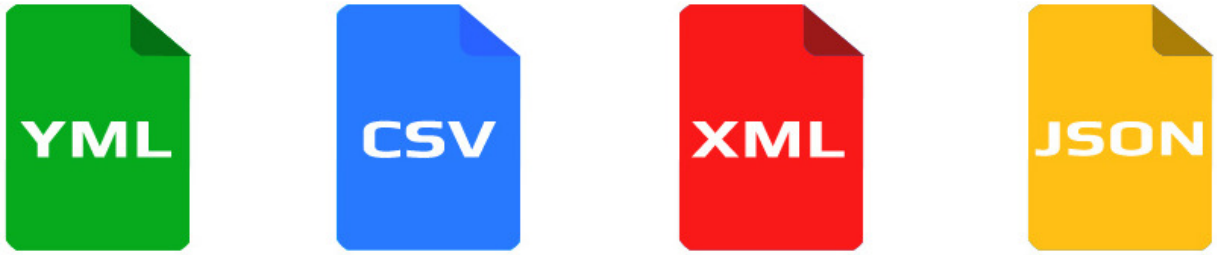
It's hard to explain puns to kleptomaniacs because they always take things literally.

The Agenda

Agenda for this session:

1. What is DATA-GEN?
2. Basic Use Example
3. DATA-GEN Options
4. Sequences
5. Writing a Generator

Some History



- 2006: XML-INTO (and XML-SAX) were added to enable reading XML documents.
  - People ask, "good for reading, but how do we write XML?"
- 2009, 2011: Various XML-INTO enhancements
  - People ask, "what about other document types? JSON-INTO? What about YAML, CSV, JSON, XDR, Property List, Pickle, OpenDDL, protobuf, OGDL, KMIP, FHIR, Feather, Arrow, EDN, CDR, Coifer, CBOR, Candle, Bond, Bencode, D-Bus, ASN.1, HOCON, MessagePack, SCaViS, Smile, Thrift, VPack?
- 2018: DATA-INTO was added, its like XML-INTO but you can get (or write) different "drivers" for different document formats. Most commonly JSON.
  - People ask, "how do we write XML and JSON?"
- 2019: DATA-GEN: Like DATA-INTO, but for generating (writing) rather than reading!

Opcode Syntax

```
DATA-GEN source-variable %DATA(result {: options}) %GEN(generator {: options});
```

- source-variable: RPG variable (usually a data structure) to generate the structured document from.
- result: Specifies the result variable, either as a character variable (default) or as an IFS pathname to write to.
- result options: Space-separated list of options that control how RPG transfers data from your source variable into the result (more to come!)
- generator: Third-party program or service program that will generate the document. The generator is what determines the format of the document you're generating.
- generator options: Character literal or RPG variable that contains options used by the generator. The format of this variable is defined by the generator program and will be different for each generator you use.

Alternate Syntax

```
DATA-GEN  *START|*END  %DATA(result {: options}) %GEN(generator {: options});
```

- *START: Used for starting a sequence of related DATA-GEN calls.
- *END: Used to end a sequence of related DATA-GEN calls

This syntax is used together with the syntax on the previous slide. You can use this to create a group of DATA-GEN calls that are used together to generate a single document.

For example, if reading a database table via SQL cursor, you may wish to start a document, then add one row at a time to structured document in a loop. To do that, use *START before the loop, a source variable inside the loop, and *END after the loop. The combined calls to DATA-GEN will be treated as a single document.

...more about this later, after I've shown you the basics...

Basic Example

Concept: Convert an RPG variable (usually a data structure) into a structured format, such as a JSON document.

```
dcl-ds address qualified;
  name   varchar(30)  inz('Scott Klement');
  street varchar(30)  inz('8825 S Howell Avenue Ste 301');
  city   varchar(20)  inz('Oak Creek');
  state  char(2)      inz('WI');
  postal varchar(10)  inz('53154');
end-ds;

dcl-s Json varchar(1000);
```

```
 DATA-GEN address %DATA(Json) %GEN('YAJLDTAGEN');
```

```
{
  "name": "Scott Klement",
  "street": "8825 S Howell Avenue Ste 301",
  "city": "Oak Creek",
  "state": "WI",
  "postal": "53154"
}
```

Requirements for DATA-GEN

DATA-GEN was:

- added to RPG in November 2019 (via PTF)
- available 7.3 and 7.4 via PTFs
- Future releases (those after 7.4) will include it at GA

NOTE: Like all RPG features released after March 2008, it will show up as a syntax error in SEU. SEU is no longer viable for anything but legacy work!!

PTF information can be found here:
http://ibm.biz/fall_2019_rpg_enhancements

Installing support for DATA-GEN will include/update the QOAR library with copybooks and sample programs from IBM

The Following Examples Use JSON

Currently, JSON is the most widely used format in REST APIs (web services)

- It has displaced XML
- Also the most popular use of DATA-GEN and DATA-INTO

Since JSON is so popular I will use it as an example for this presentation.

- YAJL is an open source JSON tool from by Lloyd Hillael. It is both very fast and very popular.
- Scott provides a YAJLDTAGEN generator (for the %GEN BIF) as part of the YAJL download from his web site
- But, do remember that DATA-GEN is not limited to JSON, it can be used for any format so long as you can find/buy/write a generator program for that format.

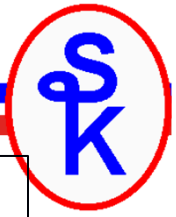
JSON Syntax

JSON is a subset of the JavaScript programming language used to represent data in JavaScript variables.

- Quoted strings represent character variables.
- Numeric literals represent numeric variables
- Special (unquoted) values of true or false represent boolean ("indicator")
- The [ ] characters represent an array elements are separated by commas.
- The { } characters represent an object (which is a data structure in RPG), with subfield names separated by their values by a colon, and subfields separated from subsequent subfields by a comma.

With that in mind, lets see that basic example again...

Basic Example -- Revisited




```
dcl-ds address qualified;
  name   varchar(30)  inz('Scott Klement');
  street varchar(30)  inz('8825 S Howell Avenue Ste 301');
  city   varchar(20)  inz('Oak Creek');
  state  char(2)      inz('WI');
  postal varchar(10)  inz('53154');
end-ds;

DATA-GEN address %DATA(Json) %GEN('YAJLDTAGEN');
```

- The { } characters represent the start/end of an object, which equivalent to an RPG data structure (represented by dcl-ds/end-ds in free format)
- The JSON subfield names are copied directly from the DS subfield names
- The values are determined by the RPG subfield contents.

```
{
  "name": "Scott Klement",
  "street": "8825 S Howell Avenue Ste 301",
  "city": "Oak Creek",
  "state": "WI",
  "postal": "53154"
}
```

%DATA Options Summary

Summary of the different options for the %DATA built-in function (BIF) – more detail on each is coming up...

- doc – controls where the document is generated string (default) or file.
- trim – remove extra blanks from strings
- countprefix – control the number of specified elements generated
- fileccsid – specifies the CCSID when creating an output file
- name – specifies the name of the top-level element (for document)
- output – should the output variable/file be cleared? Or appended?
- renameprefix – lets you specify variables containing alternate names for subfields.

```
%DATA(myStmf:'put options here')
```

DOC Option

The default is doc=string (generate results into a string)

doc=file tells DATA-GEN to write results to the IFS. The first parameter to %DATA is now the IFS path name.

Imagine the "address" example (our basic example) written to an IFS file named /home/scott/address.json

```
myStmf = '/home/scott/address.json';

data-into address %DATA(myStmf:'doc=file') %GEN('YAJLDTAGEN');
```

TRIM Option

trim=all (default)

- both leading and trailing blanks are removed from each string
- strings of interior blanks are reduced to a single blank

trim=none

- no blanks are removed
- performs the fastest

```
dcl-ds testme qualified;
  city   char(20)  inz('Oak Creek');
  state  char(2)   inz('WI');
  postal char(10)  inz('53154');
end-ds;

// {"city":"Oak Creek","state":"WI","postal":"53154"}
data-gen testme %data('trimAll.json': 'doc=file trim=all')
                %gen('YAJLDTAGEN');

// {"city":"Oak Creek           ","state":"WI","postal":"53154     "}
data-gen testme %data('trimNone.json': 'doc=file trim=none')
                 %gen('YAJLDTAGEN');
```

CountPrefix Option (1 of 3)

CountPrefix creates a prefix. Fields that use the prefix can be used to set the number (or "count") of an element to generate.

To understand, imagine you receive the following "statement.json" file from a vendor. It is a statement, telling what you owe for a given month.

```
{
  "customer": 5406,
  "stmtDate": "2018-10-05",
  "startDate": "2018-09-01",
  "endDate": "2018-09-30",
  "total": 6600.00,
  "invoices": [
     { "invoice": "99001", "amount": 1000.00, "date": "2018-09-14" },
     { "invoice": "99309", "amount": 1500.00, "date": "2018-09-18" },
     { "invoice": "99447", "amount": 500.00, "date": "2018-09-23" },
     { "invoice": "99764", "amount": 3600.00, "date": "2018-09-14" }
  ]
}
```

Now imagine the RPG code needed to generate this….

CountPrefix Option (2 of 3)

Example: countprefix=total_, then total_XYZ is the number of XYZ elements to generate.

Or, for the statement/invoice list:

```
dcl-ds statement qualified inz;
  customer  packed(4: 0);
  stmtDate  char(10);
  startDate char(10);
  endDate   char(10);
  total     packed(9: 2);
  num_invoices int(10);
  dcl-ds invoices dim(999);
    invoice char(5);
    amount  packed(9: 2);
    date    char(10);
  end-ds;
end-ds;

DATA-GEN statement
         %DATA('statement.json': 'doc=file countprefix=num_')
         %gen('YAJLDTAGEN': '{ "beautify": true }' );
```

DIM(999) specifies the maximum number of invoices we can generate -- but we won't always want 999 of them!

CountPrefix With a Zero Count

You can also use CountPrefix with a 0 count if you want to omit an element completely.

```
dcl-ds response qualified;
  success ind;
  num_errorMsg int(10);
  errorMsg varchar(100);
end-ds;

response.success = *on;          // or *OFF for 'false'
response.num_errorMsg = 0;       // or 1 if message needed

// {"success":true}      or {"success":false, "errorMsg": "message"}

data-gen response %data('errMsg.json': 'doc=file countprefix=num_')
                  %gen('YAJLDTAGEN');
```

In this example, the errorMsg field is not written to the document at all, because the num_errorMsg field is 0

FILECCSID Option

Specifies the CCSID that is used to create the output IFS file if it does not already exist.

fileccsid=utf8 (default)
- File is created as UTF-8 (CCSID 1208)

fileccsid=utf16
- File is created as UTF-16 (CCSID 1200)

fileccsid=job
- File is created in the job CCSID, or job default CCSID

fileccsid=*number*
- File is created with the specified CCSID

```
myStmf = '/home/scott/address.json';

DATA-GEN address %DATA(myStmf:'doc=file fileccsid=utf16')
                 %GEN('YAJLDTAGEN');
```

NAME Option

Controls the name assigned to the top-level element in the generator

name=(same as variable) (default)

- By default, the generator is given the name of your variable

name=(specified value)

- You can specify an alternate value

```
dcl-ds rec qualified;
  number  char(4);
  amount  packed(9: 2);
end-ds;

data-gen rec %data('nameopt.json'
                   : 'doc=file name=invoice output=continue')
             %gen('YAJLDTAGEN');
```

Example: Without the name option, it would generate a structure named 'rec' with fields named number/amount. But, due to name, it will be a structure named 'invoice'.

NOTE: Top-level JSON elements don't normally have a name, but this can be useful in a sequence (more later)

OUTPUT Option

output=clear (default -- except with *END sequence)

- Output file or variable is cleared before generating data.

output=append

- Output file or variable is appended to.

output=continue (default for *END)

- File was opened by a previous *START, and should be continued
- Required for a variable in a multi-sequence call (more later)

```
data-gen *start %data('mydoc.json': 'doc=file output=clear')
                %gen('YAJLDTAGEN');

data-gen rec    %data('mydoc.json': 'doc=file output=continue')
                %gen('YAJLDTAGEN');

data-gen *end   %data('mydoc.json': 'doc=file output=continue')
                %gen('YAJLDTAGEN');
```

RENAMEPREFIX Option

Allows you to rename json elements. For example: Suppose you wanted to create the following:

```
{
    "customer": 5406,
    "statement date": "2018-10-05",
    "start date": "2018-09-01",
    "end date": "2018-09-30",
    "statement total": 6600.00,
    "invoices": [
        { "invoice": "99001", "amount": 1000.00, "date": "2018-09-14" },
        { "invoice": "99309", "amount": 1500.00, "date": "2018-09-18" },
        { "invoice": "99447", "amount":  500.00, "date": "2018-09-23" },
        { "invoice": "99764", "amount": 3600.00, "date": "2018-09-14" }
    ]
}
```

Its not possible to create an RPG variable named "statement name", "start date", "end date" or "statement total" because RPG variables cannot have spaces in them.

(There are similar concerns with punctuation symbols, diacritics, etc.)

RENAMEPREFIX Option

The RENAMEPREFIX option lets you define fields that start with the prefix and correspond to the names of the elements. These are used to rename the output field names.

```
dcl-ds statement qualified inz;                          // {
  customer  packed(4: 0);                                //   "customer": {number},
  stmtDate  char(10);                                    //   "statement date": "{string}",
  name_stmtDate varchar(50)  inz('statement date');      //      (renames stmtDate)
  startDate char(10);                                    //   "start date": "{string}",
  name_startDate varchar(50) inz('start date');          //      (renames startDate)
  endDate   char(10);                                    //   "end date": "{string}",
  name_endDate varchar(50)   inz('end date');            //      (renames endDate)
  total     packed(9: 2);                                //   "statement total": {number},
  name_total varchar(50)     inz('statement total');     //      (renames total)
  num_invoices int(10);                                  //      (controls number of invoices)
  dcl-ds invoices dim(999);                              //   "invoices": [{
    invoice char(5);                                     //      "invoice": "{string}",
    amount  packed(9: 2);                                //      "amount": {number},
    date    char(10);                                    //      "date": "{string}"
  end-ds;                                                //   }]
end-ds;                                                  // {

DATA-GEN statement %DATA( 'statement.json'
                        : 'doc=file countprefix=num_ renameprefix=name_')
                  %gen('YAJLDTAGEN');
```

Since the RenamePrefix is name_ any field that begins with name_ is an alternative name for an element

%GEN Options

Previous slides discussed the options for the %DATA BIF. Those were handled by RPG, but there's a second place for options handled by the generator program.

```
DATA-GEN variable %DATA(document [:options])
                  %GEN( generator [:options] );
```

Options on %GEN are handled by the 3rd-party generator program and will be different for every generator program you use!

%GEN Options:

- Can be coded as a string literal. In this case, they are passed to the generator as a pointer to null-terminated (C-style) string.
- Or can be an RPG variable. In this case, the generator gets a pointer to that variable.
- By contrast, the %DATA options are always a character string.
- It is up to the parser to determine the format of the generator options and what variable type(s) it will accept.

YAJLDTAGEN %GEN Options

YAJLDTAGEN expects:

- %GEN options are passed as a small JSON document
- Must be a literal or an RPG character string variable
- No options are required – *only specify the ones you need to use.*

YAJLDTAGENs options are:

- beautify = if true, the JSON is formatted with linefeeds and indenting. (Default: false)
- escape solidus = if true, the / ("foreslash") character will be escaped. (Default: false)
- write to stdout = if true, the output JSON is written to standard output, which is used with the IBM HTTP Server (powered by Apache) to make web services. (Default: false)
- http status = numeric status code sent to the HTTP server if write to stdout is true. (Default: 200)
- sequence type = when using DATA-GEN's *START/*END sequences, this controls whether the sequences are used to build an object vs an array. (Default: "object")

```
DATA-GEN statement %DATA(myJsonVar)
                  %GEN( 'YAJLDTAGEN'
                      : '{ +
                          "beautify": true, +           // default: false
                          "escape solidus": true, +     // default: false
                          "write to stdout": true, +    // default: false
                          "http status": 200, +         // default: 200
                          "sequence type: "array" +     // default: "object"
                        }');
```

Sequences

Sequences allow you to split the document into smaller parts by calling DATA-GEN multiple times. This is ideal when building something from a database table, since it obviates the need to load all of the data into a big array before writing it.

```
data-gen *start %data('nameopt.json': 'doc=file')
                %gen('YAJLDTAGEN');

dow (some_condition);
   data-gen myDS %data('nameopt.json': 'doc=file output=continue')
                %gen('YAJLDTAGEN');
enddo;

data-gen *end   %data('nameopt.json': 'doc=file')
                %gen('YAJLDTAGEN');
```

To use sequences

1) Start a sequence with the special value *START in place of the RPG variable
2) Generate one or more variables into the sequence by calling DATA-GEN
3) Finish the sequence with the special value *END to close the file and finish generating.

Sequences, Array Example

```
**free
dcl-f PRODP disk;

dcl-c ifsFile 'product list.json';
dcl-c yajlOpts '{ +
                  "beautify": true, +
                  "sequence type": "array" +
                }';

dcl-ds prod_rec likerec(PROD:*INPUT);

setll *start PRODP;
read PRODP prod_rec;

DATA-GEN *START %DATA( ifsFile: 'doc=file')
                %GEN('YAJLDTAGEN': yajlOpts);

dow not %eof(PRODP);
   DATA-GEN prod_rec %DATA(ifsFile: 'doc=file output=continue')
                     %GEN('YAJLDTAGEN': yajlOpts );
   read PRODP prod_rec;
enddo;

DATA-GEN *END   %DATA( ifsFile: 'doc=file')
                %GEN('YAJLDTAGEN': yajlOpts);

*inlr = *on;
```

```
[
    {"PRID":5,"PPRICE":12.26,"PIMG":1,"PSTOCK":50},
    {"PRID":8,"PPRICE":12.99,"PIMG":2,"PSTOCK": 1},
    {"PRID":9,"PPRICE": 6.30,"PIMG":3,"PSTOCK":20},
    ... etc ...
]
```

Notes:

- "sequence type": "array" caused the records to form an array.
- output from a sequence must be a file
- output=continue is used to continue a sequence
- File is only closed when *END is reached

Sequences, Object Example

```
**free
dcl-f PRODP disk;

dcl-c ifsFile 'product list.json';
dcl-c yajlOpts '{ +
                  "beautify": true, +
                  "sequence type": "object" +
                }';

dcl-ds prod_rec likerec(PROD:*INPUT);

setll *start PRODP;
read PRODP prod_rec;

DATA-GEN *START %DATA( ifsFile: 'doc=file')
                %GEN('YAJLDTAGEN': yajlOpts);

dow not %eof(PRODP);
   DATA-GEN prod_rec %DATA(ifsFile: 'doc=file +
                                     name=product +
                                     output=continue')
                     %GEN('YAJLDTAGEN': yajlOpts );
   read PRODP prod_rec;
enddo;

DATA-GEN *END   %DATA( ifsFile: 'doc=file')
                %GEN('YAJLDTAGEN': yajlOpts);

*inlr = *on;
```

```
{
    "product": {
        "PRID": 5,
        "PPRICE": 12.26,
        "PIMG": 1,
        "PSTOCK": 50,
    },
    "product": {
        "PRID": 8,
        "PPRICE": 12.99,
        "PIMG": 2,
        "PSTOCK": 1,
    }
    ... etc ...
}
```

Notes:

- "sequence type": "object" caused the records to form an object
- name=product caused the name of each object field to be "product" instead of "prod_rec"
- output from a sequence must be a file
- output=continue is used to continue a sequence
- File is only closed when *END is reached

YAJLDTAGEN as a web service

YAJLDTAGEN has a special feature for writing web services:

- use this when RPG is called from Apache via ScriptAlias
- primarily for "do it yourself" style web services
- *not* for use with tools like Integrated Web Services or WebSphere

```
DATA-GEN statement %DATA(myJsonVar)
                  %GEN( 'YAJLDTAGEN'
                      : '{"write to stdout":true, "http status":200}');
```

Notes:

- `"write to stdout": true` causes the JSON to automatically be written to the Apache server which will send it back to the caller
- Despite "write to stdout", it will still be written to the output (in this case, the myJsonVar variable)
- `"http status"` lets you control the HTTP status code. Typically you'd use 200 for success, or 500 for an error.

Remember...

- ✓ It is much easier to explain DATA-GEN if I can show you examples.
- ✓ To show you examples, I need an example generator (%GEN)
- ✓ Since JSON is the most common document to use with DATA-GEN, and YAJLDTAGEN is the best JSON tool available, I used it as an example.

But DATA-GEN can be used for just about anything!

In addition to these examples, you can find:

- IBM provides GENHTMLTAB for HTML (QOAR/SAMPLE file)
- IBM provides GENPROP for property file format (QOAR/SAMPLE)
- I will also show you a CSV of my own
- *Plus whatever else you can dream up!*

Debugging the Generator

IBM provides a special environment variable to assist you with using DATA-GEN. It traces all of the information passed into the generator from your program. (Generators can add additional information as well.)

To enable it for your job:

```
ADDENVVAR ENVVAR(QIBM_RPG_DATA_GEN_TRACE) VALUE(*STDOUT)
```

Example output:

```
Start DATA-GEN
 Event 1 (StartMultiple)
End DATA-GEN
Start DATA-GEN
 Event 3 (Start)
  Event 5 (StartStruct) for prod_rec
   Event 11 (ScalarValue) for PRID
   Event 11 (ScalarValue) for PPRICE
   Event 11 (ScalarValue) for PIMG
   Event 11 (ScalarValue) for PSTOCK
```

Writing Your Own Generator

This is for the real nerds out there! (ahem, like Scott)

Imagine what you could do if you wrote your own parser!!

Why?

- Support additional document formats
- Add cool features that don't already exist!
- *BECAUSE ITS FUN!*

Ideas:

- YAML, Protocol Buffer, any other formats?
- Maybe use it to generate a spreadsheet?
- Less limiting than Open Access because not limited to a 32k flat record.

DATA-GEN Generator Overview

DATA-GEN calls the generator.

- Definitions related to the call are found in the copybook QOAR/QRPGLESRC,QRNDTAGEN
- Always one parameter, a data structure in format QrnDgParm_t that is defined in the copybook.
- The generator will be called multiple times, each time it is expected to handle a particular "event"
  - The events and their meanings are listed on the next slide.
- Each event provides one piece of the RPG variable information, and you use it to generate the document.
- The result is returned to RPG by calling subprocedures:
  - `QrnDgAddTextXXXX` writes the generated document
    - QrnDgAddText = adds UCS-2 text
    - QrnDgAddTextCCSID = adds text in any CCSID
    - QrnDgAddTextString = adds null-terminated strings
    - QrnDgAddTextNewline = adds a newline character
  - `QrnDgReportError` returns errors back to RPG
  - `QrnDgTrace` writes to the trace log

DATA-GEN Events

Event name (constant)	Description
QrnDgEvent_01_StartMultiple	Start of a sequence (*START was used)
QrnDgEvent_02_EndMultiple	End of a sequence (*END was used)
QrnDgEvent_03_Start	Start of the variable to create document from
QrnDgEvent_04_End	End of the variable to create document from
QrnDgEvent_05_StartStruct	Start of a data structure
QrnDgEvent_06_EndStruct	End of a data structure
QrnDgEvent_07_StartScalarArray	Start of an array that is not a data structure array
QrnDgEvent_08_EndScalarArray	End of an array that is not a data structure array
QrnDgEvent_09_StartStructArray	Start of a data structure array
QrnDgEvent_10_EndStructArray	End of a data structure array
QrnDgEvent_11_ScalarValue	A single variable value (subfield)
QrnDgEvent_12_Terminate	Terminate process (if doTerminateEvent was set to '1')

State & Cleanup

The parameter contains a special field named generatorState. It is a pointer that you can set to any memory address you wish.

DATA-GEN will retain this pointer across all calls to your generator program, so this can be used to provide an area of memory that's available to all generator events.

The QrnDgEvent_12_Terminate event is only called if you set the doTerminateEvent indicator in the parameter. If this field is set, it will always be called at the very end. You cannot use the QrnDgXXXX procedures during a QrnDgEvent_12_Terminate event.

This terminate event is perfect for cleaning up the pointer you've assigned to the generatorState.

Writing a Generator: CSV Example

This example:

- Name is CSVGEN
- Builds a CSV record from the fields in a DS
- Use a sequence (*START/*END) or an array to build a whole file
- Example of the flow of events
- Example of extracting values from fields

Writing a parser is a more "systems" style of programming using pointers, APIs, etc. This requires you to channel your inner nerd.

CSV File Format

This file represents a list of products

- Product id
- Product name
- Other stuff like price, quantity, etc.

Format of Comma Separated Values (CSV) file:

- Stored in IFS (not a database)
- Character fields are in quotes, numbers are not
- Each record is terminated by a "new line" character

```
5,"NALGENE CANTEEN 48 OZ","",12.26,1,50,25,1,2,"Y"
8,"NALGENE CANTEEN 96 OZ","",12.99,2,1,25,1,1,""
9,"NALGENE 16 OZ WIDE-MOUTH LEXAN","",6.30,3,20,25,1,1,""
10,"NALGENE 32 OZ WIDE-MOUTH LEXAN","",8.10,4,66,25,1,1,"Y"
11,"NALGENE WIDE MOUTH LOOP-TOP BO","",7.98,5,57,25,1,1,"Y"
13,"MOTOROLA PEANUT RADIO MODEL T6","",250.00,7,75,33,1,1,""
14,"MOTOROLA PEANUT RADIO MODEL T6","",115.00,8,20,78,0,0,"Y"
15,"MOTOROLA PEANUT RADIO MODEL T6","",146.00,9,37,33,0,0,""
```

Plan for CSV Generator


```
dcl-ds record qualified;
   Product_Id packed(7: 0);
   Product_Name char(30);
   // ... other fields here ...
end-ds;
```

Events that will occur:

- QrnDgEvent_03_Start = start of entire document
- QrnDgEvent_05_StartStruct = start of DS (name = "record")
  - Generator should: Start a new CSV record (clear the record)
- QrnDgEvent_11_ScalarValue = field (name="Product_Id")
  - Generator should: Add the field to the record
- QrnDgEvent_11_ScalarValue = field (name="Product_Name")
  - Generator should: Add the field to the record
- QrnDgEvent_06_EndStruct = end of DS
  - Generator should: Send the record + newline back to DATA-GEN
- QrnDgEvent_04_End = end of entire document
- QrnDgEvent_12_Terminate = clean up memory/files

How To Handle Multiple Rows?

```
dcl-ds record qualified;
   Product_Id packed(7: 0);
   Product_Name char(30);
   // ... other fields here ...
end-ds;

*START, record, record, *END
```

```
dcl-ds all_records qualified;
   num_records int(10);
   dcl-ds records dim(9999);
      Product_Id packed(7: 0);
      Product_Name char(30);
      // ... other fields ...
   end-ds;
end-ds;
```

The previous slide ("the plan") showed only one record.

- We need to handle many
- Maybe it could be an array, and countprefix could be used to control how many are written
- Alternately, maybe *START/*END could be used (a sequence)

In either case, the process would still use the same plan – the code would just be repeated for each record.

What would that look like from the calling program's perspective?

Calling the CSV Generator (Array)

The CSV generator can be called to build a CSV file from an array.
For example:

```
**free
dcl-f PRODP disk;

dcl-ds prodrec likerec(PROD:*INPUT) inz;

dcl-ds prodlist qualified;
   num_item int(10) inz(0);
   item likeds(prodrec) dim(9999) inz(*likeds);
end-ds;

setll *start PRODP;
read PRODP PRODREC;

dow not %eof(PRODP);
   prodlist.num_item += 1;
   prodlist.item(prodlist.num_item) = prodrec;
   read PRODP PRODREC;
enddo;

data-gen prodlist %data( 'item list.csv'
                       : 'doc=file countprefix=num_')
                 %GEN('CSVGEN');

*inlr = *on;
```

Database records are loaded into the 'prodrec' data structure.

CSV data will be generated from this array of 'prodrec' DSes. There will be one row in the CSV per record in the array. The fields in the CSV will match those of the 'prodrec'

DATA-GEN calls the 'CSVGEN' generator. It handles the work of creating a CSV file from the DS

Calling the CSV Generator (Sequence)

Alternately, the CSV generator can be called using a sequence. This way, we only need one database row in memory at any time

```
**free
dcl-f PRODP disk;

dcl-ds prodrec likerec(PROD:*INPUT) inz;

setll *start PRODP;
read PRODP PRODREC;

data-gen *START  %DATA( 'product list.csv' : 'doc=file')
                 %GEN('CSVGEN');

dow not %eof(PRODP);
   data-gen PRODREC %DATA( 'product list.csv'
                         : 'doc=file output=continue')
                    %GEN('CSVGEN');
   read PRODP PRODREC;
enddo;

data-gen *END   %DATA( 'product list.csv': 'doc=file')
                 %GEN('CSVGEN');

*inlr = *on;
```

Database records are loaded into the 'prodrec' data structure.

***START begins the CSV file**

Each row is written using the PRODREC DS

***END completes the CSV sequence**

CSV Generator (1 of 7)

The goal of this generator will be to create CSV file like this:

```
ctl-opt OPTION(*SRCSTMT:*NODEBUGIO:*NOSHOWCPY)
        main(CSVGEN)
        CCSID(*UCS2 : *UTF16)
        DFTACTGRP(*NO)
        ACTGRP('CSVGEN')
        COPYRIGHT('Copyright 2020 Scott Klement');

/copy QOAR/QRPGLESRC,QRNDTAGEN

dcl-ds state_t qualified template;
   start ind;
   data  varucs2(65535);
end-ds;

dcl-proc CSVGEN;

   dcl-pi *N;
      parm likeds(QrnDgParm_t);
   end-pi;

   pQrnDgEnv = parm.env;
```

RPG linear main. Eliminates the RPG cycle by pointing to a procedure as the "main procedure"

We'll use this data structure as our "generatorState" to track the state of the generator between calls.

DATA-GEN always passes just one parameter.

It is a DS, defined in the (IBM supplied) QRNDTAGEN copybook

To allow for thread safety, the subprocedures we call are based on a pointer. This pointer is defined in the copybook and must be set on each call.

CSV Generator (2 of 7)

The goal of this parser will be to read a CSV file like this:

```
dcl-s  p_state pointer;
dcl-ds state likeds(state_t) based(p_state);

parm.doTerminateEvent = *ON;

if parm.generatorState = *null;
  parm.generatorState = %alloc(%size(state_t));
  p_state = parm.generatorState;
  state.start = *on;
  state.data  = '';
endif;

p_state = parm.generatorState;
```

Set up the state data structure and a doTerminateEvent flag that we'll use to clean it up.

By using allocated memory, we can ensure a separate copy for each instance of DATA-GEN that runs.

CSV Generator (3 of 7)

Remember: Our generator is called many times (in a loop) and is called separately for different things that are found in the RPG variable

```
select;
  when parm.event = QrnDgEvent_05_StartStruct;

     startRow(parm: state);

  when parm.event = QrnDgEvent_06_EndStruct;

     endRow(parm: state);

  when parm.event = QrnDgEvent_11_ScalarValue;

     addField(parm: state);

  when parm.event = QrnDgEvent_12_Terminate;

     dealloc parm.generatorState;
     p_state = *null;

  endsl;
```

At the beginning of a data structure, we want to start a new CSV row. So call the startRow() procedure.

At the end of a data structure, we'll write the row to the CSV file (via DATA-GEN)

ScalarValue is a simple (not DS or array) subfield value. This means a new field should be added to the CSV file

Because we set the doTerminateEvent flag to '1', the terminate event will be called at the end. It'll be used to clean up the allocated memory.

CSV Generator (4 of 7)

When a new data structure begins, all we need to do is start a fresh row in the CSV file. This is done by blanking out all of the data, and setting the 'start' flag *ON.

```
dcl-proc startRow;

  dcl-pi *n;
    parm likeds(QrnDgParm_t);
    state likeds(state_t);
  end-pi;

  state.start = *on;
  state.data  = '';

end-proc;
```

CSV Generator (5 of 7)

Each field in the DS is added as a field in the CSV row. RPG passes the field data in the 'scalar' parameter of the parm. The data is provided as a pointer to a series of UCS-2 characters.

```
dcl-proc addField;

  dcl-pi *n;
    parm  likeds(QrnDgParm_t);
    state likeds(state_t);
  end-pi;

  dcl-s p_buf   pointer;
  dcl-s buf     ucs2(65535) based(p_buf);
  dcl-s tempVal varucs2(65535);

  if parm.scalar.valueLenChars <= 0;
     tempVal = '';
  else;
     p_buf = parm.scalar.value;
     tempVal = %subst(buf:1:parm.scalar.valueLenChars);
  endif;

  if state.start = *on;
     state.start = *off;
  else;
     state.data += %ucs2(',');
  endif;
```

If no characters provided, this field is blank. If characters were provided, use a pointer and %SUBST to get the chars

If we're not at the start of the row, add a comma to the row before adding the data.

CSV Generator (6 of 7)

```
 select;
  when parm.scalar.dataType = QrnDatatype_Decimal
    or parm.scalar.dataType = QrnDatatype_Integer
    or parm.scalar.dataType = QrnDatatype_Unsigned
    or parm.scalar.dataType = QrnDatatype_Float;

     state.data += tempVal;

  other;

     state.data += %ucs2('"') + tempVal + %ucs2('"');

  endsl;

end-proc;
```

Numeric fields are added to the CSV without any surrounding quotes

Anything else (character, indicator, date, time, timestamp, etc) will be surrounded by quotes

CSVGEN is called for each field, so the above code will be repeated for each field in the data structure. When done, 'state.data' will contain a comma separated list of all of the field values.

CSV Generator (7 of 7)

```
dcl-proc endRow;

  dcl-pi *n;
    parm  likeds(QrnDgParm_t);
    state likeds(state_t);
  end-pi;

  if state.start = *off;

    QrnDgAddText( parm.handle
                : %addr(state.data:*DATA)
                : %len(state.data) );

    QrnDgAddTextNewLine( parm.handle );

    state.start = *on;
    state.data  = '';
  endif;

end-proc;
```

Check to see if at least one field was added.

Add the row, followed by a "new line" character to the CSV document by calling DATA-GEN's QrnDgAddText routines.

QrnDgAddText and QrnDgAddTextNewLine report our data to DATA-GEN. DATA-GEN itself will take care of writing the result to the IFS.

More Information

PTF information for DATA-GEN support on IBM i 7.3 and 7.4
http://ibm.biz/fall_2019_rpg_enhancements

IBM's *Writing a Generator for the RPG DATA-GEN Operation Code:*
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/rzasm/roaDataGen.htm

DATA-GEN operation code in the *ILE RPG Reference Manual*.
https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_74/rzasd/zzdatagen.htm

From Scott Klement:

Scott's IBM i Port of YAJL (includes YAJLDTAGEN)
https://www.scottklement.com/yajl/

Scott's CSVutil (includes CSVGEN):
https://www.scottklement.com/csv/

This Presentation

You can download a PDF copy of this presentation

http://www.scottklement.com/presentations/

Thank you!