

Overloading



Subprocedures in RPG

Presented by

Scott Klement

<http://www.scottklement.com>

© 2020-2023, Scott Klement

Debugging [*de-buhg-ing*] -verb

1. Being the detective in a crime movie where you are also the murderer.

What is Overloading?



Overloading allows a language (such as RPG) to select between multiple routines based on the parameter list.

```
X = MyRoutine( charVariable );  
X = MyRoutine( numVariable );
```

Example: Two different routines both called as MyRoutine(), but they have different parameter lists.

Why?

- Sometimes the same thing might be done with different ways, such as retrieving a customer by their name vs. their account number.
- Naming them the same thing means callers only have to remember one name.
- Since the names are the same, RPG determines which routine to call based on its parameter list.
- Therefore, the parameter lists must be different.

Overload: Basic Example



.....1.....2.....3.....4.....5.....6.....7.....8

```
dcl-proc format_date;
  dcl-pi *n varchar(100);
    dateParm DATE CONST;
  end-pi;

  return %char(dateParm:*ISO);
end-proc;

dcl-proc format_time;
  dcl-pi *n varchar(100);
    timeParm TIME const;
  end-pi;

  return %char(timeParm:*HMS:);
end-proc;

dcl-proc format_message;

  dcl-pi *n VARCHAR(100);
    msgid          CHAR(7)          CONST;
    replacement_text VARCHAR(100)  CONST OPTIONS(*NOPASS);
    message_file   char(20)         CONST OPTIONS(*NOPASS);
  end-pi;

  // code to look up message in message file and return it
```

Three different procedures that format strings.

format_date formats a date into a string

format_time formats a time into a string

format_message looks up a message in a message file, formats it, and returns a string.

It would be nice if they could all be called just "format".

6

Overload: Basic Example



.....1.....2.....3.....4.....5.....6.....7.....8

```
DCL-PR format VARCHAR(100)  OVERLOAD( format_time
                                     : format_date
                                     : format_message);

DCL-S result varchar(50);

result = format(%date());           // 1
result = format(%time());           // 2
result = format('CPF2105' : 'PRDMASTER QUSRSYS FILE '); // 3
```

Using overloading all can be called as "format"

- #1 calls **format_date** because the parameter is a date
- #2 calls **format_time** because the parameter is a time
- #3 calls **format_message** because there are 2 character parameters. (the third parameter had options(*nopass) so was optional.)

7

Overloading: Rules & Limitations



```
.....1.....2.....3.....4.....5
DCL-PR format_date VARCHAR(100);
    dateParm DATE CONST;
END-PR;

DCL-PR format_time VARCHAR(100);
    timeParm TIME CONST;
END-PR;

DCL-PR format_message VARCHAR(100);
    msgid CHAR(7) CONST;
    rpltext VARCHAR(100) CONST OPTIONS(*NOPASS);
    msgfile CHAR(20) CONST OPTIONS(*NOPASS);
END-PR;

DCL-PR format VARCHAR(100)
    OVERLOAD( format_time
              : format_date
              : format_message);
```

Overloading Rules:

- An overload prototype has **no corresponding END-PR**
- The OVERLOAD keyword provides a list of "candidate" prototypes. (These must be defined elsewhere.)
- All candidates must have the **same return type**
- For each call, RPG will try to call each prototype in the list. It will use the **first one that could be called** (according to the compiler's rules for calling a prototype)
- Remember: Prototypes allow **character variables larger** than those defined, and keywords like **CONST** and **VALUE** can allow a prototype to be used even if the parameters do not match exactly.

```
// 'CPF2105' is character, so would give an error calling format_time
// 'CPF2105' is character, so would give an error calling format_date
// ...but... 'CPF2105' and 'PRDMASTER...' would be valid parameters for format_message.

result = format('CPF2105' : 'PRDMASTER QUSRSYS FILE');
```

Business Example (1 of 3)



The "format" example was for formatting a string, something less common in a business example. For this example, I'll show something more common in a business scenario: retrieving a customer record.

```
.....1.....2.....3.....4.....5.....6.....7.....8
**free
ctl-opt dftactgrp(*no) option(*srcstmt);

dcl-ds CUSTOMER_T ext extname('CUSTFILE':*INPUT) qualified end-ds;
dcl-ds cust likeds(CUSTOMER_T);

DCL-PR getCustomer likeds(CUSTOMER_T) OVERLOAD( getCustomer_num
                                                : getCustomer_name );

cust = getCustomer('Acme Foods');
cust = getCustomer( 504 );
cust = getCustomer('Fleming Co');
cust = getCustomer( 495 );
cust = getCustomer(123);
cust = getCustomer('Wonky Cust Name');

*inlr = *on;
```

As you can see, getCustomer can retrieve a record by the customer's name or account number.

Business Example (2 of 3)



The OVERLOAD keyword listed two prototypes, `getCustomer_num` and `getCustomer_name`. Here is the code for `getCustomer_num`.

```
.....1.....2.....3.....4.....5.....6.....7.....8

dcl-proc getCustomer_num;

  dcl-pi *n likeds(CUSTOMER_T);
    inCustNo packed(5: 0) value;
  end-pi;

  dcl-f CUSTFILE disk keyed static;
  dcl-ds CUSTREC likeds(CUSTOMER_T);

  chain inCustNo CUSTFILE CUSTREC;
  if not %found;
    reset CUSTREC;
  endif;

  return CUSTREC;

end-proc;
```

11

Business Example (3 of 3)



The OVERLOAD keyword listed two prototypes, `getCustomer_num` and `getCustomer_name`. Here is the code for `getCustomer_name`.

```
.....1.....2.....3.....4.....5.....6.....7.....8

dcl-proc getCustomer_name;

  dcl-pi *n likeds(CUSTOMER_T);
    inName varchar(30) const;
  end-pi;

  dcl-f CUSTFILEL1 disk keyed static;
  dcl-ds CUSTRECL1 likerec(CUSTFILEF:*INPUT);
  dcl-ds CUSTREC likeds(CUSTOMER_T);

  chain (inName) CUSTFILEL1 CUSTRECL1;
  if not %found;
    reset CUSTREC;
  else;
    eval-corr CUSTREC = CUSTRECL1;
  endif;

  return CUSTREC;

end-proc;
```

12

Overloading: Problem (1 of 2)



The getCustomer example worked because a character string can't be passed to a numeric parameter, and vice-versa.

But what if I wanted to look up a customer a different way that also took a character parameter?

```
.....1.....2.....3.....4.....5

dcl-proc getCustomer_contact;

  dcl-pi *n likeds(CUSTOMER_T);
    inContact varchar(30) const;
  end-pi;

  dcl-f CUSTFILEL2 disk keyed static;
  dcl-ds CUSTRECL2 likerec(CUSTFILEF:*INPUT);
  dcl-ds CUSTREC likeds(CUSTOMER_T);

  chain (inContact) CUSTFILEL2 CUSTRECL2;
  if not %found;
    reset CUSTREC;
  else;
    eval-corr CUSTREC = CUSTRECL2;
  endif;

  return CUSTREC;

end-proc;
```

This example looks up a customer by their contact name (vs. the name of the business)

Notice that getCustomer_contact and getCustomer_name accept a character parameter.

What will happen?

13

Overloading: Problem (2 of 2)



The getCustomer example worked because a character string can't be passed to a numeric parameter, and vice-versa.

But what if I wanted to look up a customer a different way that also took a character parameter?

```
.....1.....2.....3.....4.....5.....6.....7.....8

000107  DCL-PR  getCustomer likeds(CUSTOMER_T)
                                OVERLOAD( getCustomer_num
                                           : getCustomer_name
                                           : getCustomer_contact );
.
.
000111  cust = getCustomer('Acme Foods');
000112  cust = getCustomer( 504 );
000113  cust = getCustomer('George Smith');
.
.
*RNF0203 30      000111  PROTOTYPES GETCUSTOMER_NAME AND GETCUSTOMER_CONTACT BOTH
                                MATCH THE CALL; FIRST MATCH ASSUMED.
*RNF0203 30      000113  PROTOTYPES GETCUSTOMER_NAME AND GETCUSTOMER_CONTACT BOTH
                                MATCH THE CALL; FIRST MATCH ASSUMED.
```

Oops, now neither of them works! The compile fails with the **Sev 30** error.

14

OPTIONS(*EXACT)



OPTIONS(*EXACT) is a keyword placed on a parameter in a prototype.

Without **OPTIONS(*EXACT)**

- You can pass character variables longer than the definition on the prototype
- You can pass different data structures than the one shown on the prototype.

With **OPTIONS(*EXACT)**

- The size of character strings must match
- A data structure must be related via LIKEDS to the same data structure defined on the prototype.

16

OPTIONS(*EXACT) Example



Without OPTIONS(*EXACT) you can pass variables longer than the prototype.
With OPTIONS(*EXACT) they must match.

```
.....1.....2.....3.....4.....5.....6.....7.....8

dcl-pr p1;
  parm5 char(5);
end-pr;

dcl-pr p2;
  parm5Exact char(5) OPTIONS(*EXACT);
end-pr;

dcl-s fld10 char(10) inz('abcdefghij');

p1 (fld10); // Works! You can pass CHAR(10) for CHAR(5)
p2 (fld10); // Error! Can't pass CHAR(10) to char(5) due to *EXACT
```

17

OPTIONS(*EXACT) Example



With OPTIONS(*EXACT) a data structure must be related via LIKEDS to the one on the prototype, or it will not be allowed.

```
.....1.....2.....3.....4.....5.....6.....7.....8

dcl-ds Name qualified;
  name varchar(30);
end-ds;
dcl-ds Contact qualified;
  contact varchar(30);
end-ds;

dcl-pr p1;
  myParm likeds(Name) const options(*exact);
end-pr;

p1(Name);    // Works!  Name matches the LIKEDS.

p1(Contact); // Fails with the error message below because its not
              // related to 'Name' via LIKEDS - it doesn't matter that
              // Name & Contact share the same data types!

*RNFO203 THE TYPE AND ATTRIBUTES OF THE VALUE OR CONST PARAMETER 1
          DO NOT MEET THE OPTION(*EXACT) REQUIREMENTS.
```

18

*EXACT with Overloading



To Review:

- OVERLOAD tries each prototype listed to find one that works.
- OPTIONS(*EXACT) "tightens" parameter checking on a prototype.

Therefore, if OVERLOAD and *EXACT are used together:

- Different data structures let us differentiate different candidate prototypes.
- Different size character fields can also be used to differentiate candidate prototypes.

19

Overload with *EXACT Example



```
.....1.....2.....3.....4.....5
dcl-ds Contact_t qualified template;
  data varchar(30);
end-ds;
.
.
dcl-proc getCustomer_contact;

  dcl-pi *n likeds(CUSTOMER_T);
  inContact likeds(Contact_t) const options(*exact);
end-pi;

  dcl-f CUSTFILEL2 disk keyed static;
  dcl-ds CUSTRECL2 likerec(CUSTFILEF:*INPUT);
  dcl-ds CUSTREC likeds(CUSTOMER_T);

  chain (inContact) CUSTFILEL2 CUSTRECL2;
  if not %found;
    reset CUSTREC;
  else;
    eval-corr CUSTREC = CUSTRECL2;
  endif;

  return CUSTREC;
end-proc;
```

To solve the contact name problem, you can use *EXACT with LIKEDS.

1. Define a DS template (usually in a copy book)
2. Use LIKEDS on the prototype and PI
3. Use OPTIONS(*EXACT)
4. Call it with the data structure

Thanks to options(*EXACT) it will now work!

```
.....1.....2.....3.....4.....5
dcl-ds Contact likeds(Contact_t);
.
.
Contact.data = 'George Smith';
cust = getCustomer(Contact);
```

20

Not Just for ILE Procedures



The preceding examples have shown ILE subprocedures (both local and in a service program.)

Subprocedures are the most common place where overloading is used...

... but OVERLOAD can be used for any type of prototype.

- Procedures (with or without EXTPROC) as shown in earlier examples
- EXTPGM = external program (program calls)
- EXTPROC(*JAVA) = calls to Java methods from RPG

You can even mix different types of calls in one overload statement!

The same rules apply:

- All must have the same return type (or no return value)
- RPG will try each one, and use the first one that's allowed

27

Subprocedure / Program Mix (1 of 3)



To demonstrate this, I'll do the following:

Create an overloaded prototype called CMDEXEC (Command Execute)

1. Able to call the IBM-supplied QCMDEXC API. (This is a program call)
2. If no length is provided, call a subprocedure that calculates the length, but then calls QCMDEXC. (Subprocedure call)

```
.....1.....2.....3.....4.....5.....6.....7.....8
dcl-pr QCMDEXC_ibm extpgm('QCMDEXC');
  command char(32702) const;
  length  packed(15: 5) const;
  igc    char(3)      const options(*nopass);
end-pr;

dcl-pr CMDEXEC_NoLen;
  command varchar(32702) const;
end-pr;

dcl-pr CMDEXEC OVERLOAD( QCMDEXC_ibm
                        : CMDEXEC_NoLen );
```

EXTPGM means this is a program call

Without EXTPGM, it is a procedure call

Both are used in one OVERLOAD keyword.

28

Subprocedure / Program Mix (2 of 3)



The preceding slide was a copy book named CMDEXEC_H.

This is a code for the accompanying service program (for the subprocedure call)

```
.....1.....2.....3.....4.....5.....6.....7.....8
**free

ctl-opt nomain option(*srcstmt);

/copy CMDEXEC_H

dcl-proc CMDEXEC_NoLen export;

  dcl-pi *N;
    command varchar(32702) const;
  end-pi;

  QCMDEXC_ibm( command: %len(command) );

end-proc;
```

29

Subprocedure / Program Mix (3 of 3)



This program (I named it OVERLOAD7) calls the CMDEXC prototype:

- When a length is passed, it calls the IBM QCMDEXC
- Otherwise, calls the subprocedure to calculate the length.

```
.....1.....2.....3.....4.....5.....6.....7.....8
**free

ctl-opt option(*srcstmt: *noshowcpy);

/copy CMDEXEC_H

dcl-s cmd varchar(500);

cmd = 'DSPMSG QSYSOPR';
CMDEXEC(cmd: %len(cmd)); // calls standard API

CMDEXEC('DSPMSG QSYSOPR'); // no length, so calls CMDEXEC_NoLen

*inlr = *on;
```

30

This Presentation



You can download a PDF copy of this presentation, as well as the accompanying code:

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

Thank you!

34