Parameters and Prototypes

Presented by

Scott Klement

http://www.scottklement.com

“There are 10 types of people in the world.
Those who understand binary, and those who don’t.”

Who are you?

Scott Klement’s qualifications:

- Klement Sausage Co, Inc.
  - IT Manager and Senior Programmer
  - http://www.klements.com
- System iNEWS magazine
  - Technical Editor (also, author)
  - http://www.iseriesnetwork.com
- System iNetwork Programming Tips
  - e-Newsletter Editor
  - http://www.iseriesnetwork.com/provipcenter/
- Speaker
  - User Groups, COMMON, and RPG Summit
- Award Winner
  - Recipient of a 2005 iSeries Innovation Award (by IBM and COMMON)
  - Recipient of the 2005 Gary Guthrie Award for Excellence in Technical Writing (by System iNEWS)
  - ASBPE Awards 2006 Western Region Silver Medalist for Feature Series (RPG and the IFS)
  - COMMON Speaker of Merit

Why talk about parameters?

There are many reasons that parameters are an important tool for today’s programmer.

- Parameters are the cornerstone of modern programming!
- Without parameters, ILE is nothing.
- Without parameters, Object-Oriented code doesn’t work.
- They are much more versatile than older techniques like the LDA.
- ***Parameters are more important today than ever before!***
- ***Too many System i programmers don’t understand how parameters work!***
- There are some recent features that are worth learning.

Two Way Parameters (1 of 2)

Parameters between programs are more valuable in i5/OS than they are on a Windows or Unix system because they let you pass data both ways. You can use them to supply input values, but you can also use them to return information.

On other systems, they're input-only.

The two-way parameter is achieved using "shared memory".

When one program calls another, the ***only*** thing that’s passed between them is an address in the computer’s memory where the parameter starts. Nothing else is passed.

- Allows two-way.
- Is very efficient (only 16 bytes have to be passed)

Two Way Parameters (2 of 2)

Your computer's memory is shared by everything running on it, so the operating system has to keep track of which spaces are in use, and which ones are available.

```
PGM
  DCL VAR(&MYNBR) TYPE(*DEC) LEN(5 0)
  CHGVAR VAR(&MYNBR) VALUE(54321)
  CALL PGM(TESTPGM) PARM(&MYNBR)
ENDPGM
```

The DCL statement asks the OS for 3 bytes of memory. The OS replies with an "address 1000".

The PARM statement tells TESTPGM that there's one parameter, and that it's in location 1000.

997	*998*	*999*	**1000**	**1001**	**1002**	*1003*	*1004*	*1005*
			0	2	4			
			1	3	F			

```
PGM PARM(&COOLNUM)
    DCL VAR(&COOLNUM) TYPE(*DEC) LEN(5 0)
    CHGVAR VAR(&COOLNUM) VALUE(1234)
ENDPGM
```

The &COOLNUM variable is put in address 1000 because it's in the space provided for parameter one.

Since the first program is still referencing area 1000, it sees the new value.

What about the command line?

If parameters are passed by sharing the address of the variables, what happens

- When you call from a command line, where there aren't variables?
- When you pass a literal on the CALL statement?
- When you use an API like QCMDEXC where all the parameters are together in one variable?

```
CALL PGM(TESTPGM) PARM(18)
CALL PGM(TESTPGM) PARM('WONKAVISION')
```

- The operating system creates temporary variables for your parameters.
- It passes the addresses of those temporary variables.
- Since you didn't specify any variable size, it makes one up according to these rules:
  1. Numeric variables are always "packed" (*DEC) and 15,5
  2. Character variables are 32 chars long, and padded with blanks
  3. If a character variable is more than 32 bytes, the exact length of the parameter value is used.

Command Line Examples (1/2)

Remember, it will ask the operating system for memory, just as a variable did.

```
CALL PGM(TESTPGM) PARM(18)



CALL PGM(TESTPGM) PARM('HELLO')



CALL PGM(TESTPGM) PARM('A VERY VERY VERY VERY
VERY LONG STRING')
```

Numbers will be 15,5

(Positions 1000-1007)

This string is 5 chars long, so QCMD will ask for 32 characters, the first 5 will be HELLO, the remaining 27 will be blank.

(Pos 1000-1031)

This string is 38 chars long, and so will be a 38 character parameter with no padding.

(Pos 1000-1037)

Command Line Examples (2/2)

```
PGM PARM(&MSG)
    DCL VAR(&MSG) TYPE(*CHAR) LEN(30)
    SNDMSG MSG(&MSG) TOUSR(QSYSOPR)
ENDPGM



PGM PARM(&MSG)

    DCL VAR(&MSG) TYPE(*CHAR) LEN(80)

    SNDPGMMSG MSGID(CPF9897) TOMSGQ(*EXT) +

              MSGTYPE(*STATUS) MSGDTA(&MSG)
ENDPGM
```

This'll work from the command line, since 30 is less than 32.

This might be a problem, since 80 is more than 32. You have to type at least 80 characters (not including trailing spaces) or you'll be viewing memory that's not part of what was passed from the command line.

Look Out, It’s a Trick!

```
FQSYSPRT   O    F  132         PRINTER

D Data            ds
D   Name                       10A
D   Address                    30A

c                   call      'GETNAME'
c                   parm                    Name

c                   except
c                   eval      *inlr = *on

OQSYSPRT   E
O                                              'Name='
O                       Name
O                                           +3 'Address='
O                       Address
```

Position 1000-1009

Position 1010-1039

```
Name=Scott C Kl    Address=ement
```

```
D Name            s            15A

C     *ENTRY        PLIST
C                   PARM                    Name

C                   eval      Name = 'Scott C Klement'
c                   return
```

Position 1000-1014

Like a Data Structure?

A data structure isn’t actually used by the operating system. However, thinking of it this way might make it easier to understand. Think of your computer’s memory as one big data structure (billions of bytes long!)

```
D MainStorage      ds

       .... lots of other stuff here....

D   pgm1_data              1000    1039

D   pgm1_name              1000    1009

D   pgm1_address           1010    1039


D   pgm2_name              1000    1014

       .... lots of other stuff here....
```

The Problem

I deliberately used a data structure for name and address so I could control the memory that followed the name parameter. What if I hadn't done that? What would've been in positions 1010-1014?

- Maybe unused memory (problem goes unnoticed!)
- Maybe another variable in my program.
- Maybe a variable in another program!
- Maybe a variable used by the operating system!
- Maybe memory that I'm not allowed to use!

WHY DIDN'T IT WARN ME?

How could it? Each program doesn't know how the other program works! They can't read each other's code… Remember, the only thing they pass to one another is an address!

The Solution

The solution is to code the "GETNAME" program with a program interface and prototype.

A Program/Procedure Interface (PI) is:

- Like an **`*ENTRY PLIST`** *(but better!)*
- Requires a matching prototype to work.
- The replacement for **`*ENTRY PLIST`** in free-format.

A Prototype (PR) is:

- A "blueprint" for making a call.
- It contains the name of the program to be called.
- It tells the compiler which parameters that program needs.
- The compiler can then make sure that the parms match.

The prototype helps make the calling of a program self-documenting.

A prototype also adds a lot of "convienience" functionality, as I'll demonstrate in a bit. All of IBM's new functionality related to parms since V3R2 has gone into prototypes!

Saved by the Prototype

One member for the prototype (SOURCELIB/PROTOTYPE,GETNAME)

```
D GetName         PR                  ExtPgm('GETNAME')
D   name                         15A
```

The prototype must match the Program Interface (PI) in the program:

```
 /copy sourcelib/prototypes,getname
D GetName         PI
D   Name                         15A

C                   eval      Name = 'Scott C Klement'
c                   return
```

If the caller uses the prototype, it'll protect him from mistakes:

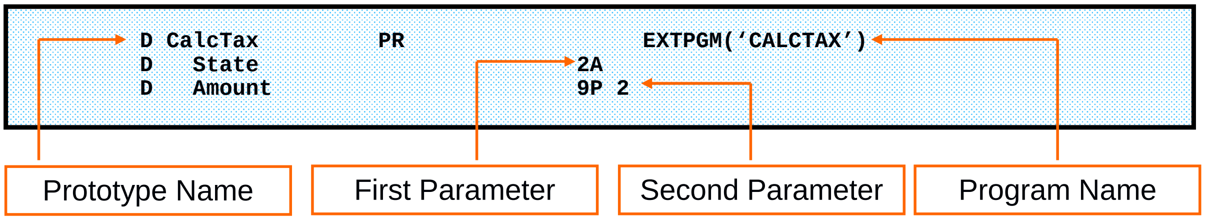
```
 /copy sourcelib/prototypes,getname

D Data            ds
```

RNF7535 The type and attributes of parameter 1 do not match those of the prototype.

```
  .
c                   callp     GetName( Name )
```

Prototypes for Programs

A prototype is very much like a parameter list (PLIST), but is newer and has a lot of additional features. You can use a prototype to call a program, a subprocedure, or a Java class.



```
D CalcTax         PR                  EXTPGM('CALCTAX')
D   State                        2A
D   Amount                       9P 2
```

- **Prototype name**

  This is the name you'll use when using the prototype to make a call. By default, it's also the name of the subprocedure that it calls. Add EXTPGM to make it call a program.

- **First Parameter**

  The first parameter to the procedure (name is for documentation, no variable is declared.)

- **Second Parameter**

  You can have as many parameters as you like, from 0-255 to a program, or 0-399 to a procedure.

- **External Program Name**

Calling Older Programs

You can use prototypes to call RPG III programs, RPG IV programs that still use *ENTRY PLIST, or even programs written in other languages (CL, COBOL, C).

```
D GetIp           PR                  ExtPgm('GETIP')
D   Device                      10A
D   Address                     15A

D MyDev           s             10A
D MyAddr          s             15A

 /free
      MyDev = 'DSP01';
      callp GetIp( MyDev : MyAddr );
 /end-free
```

You only need a PI for input (*ENTRY PLIST) parameters, not when calling something else.

That'll work even though GETIP is a CL program. It would also work if GETIP was an RPG program that used *ENTRY PLIST (in RPG III or RPG IV).

Introducing CONST

When you specify CONST, the compiler won't let you change the value of the parameter during the call.

```
FPRICELIST IF   E           K DISK

 /copy prototypes,getPrice

D GetPrice        PI
D   ItemNo                       5P 0 const
D   Zone                         1A   const
D   Price                        9P 2

 /free
   chain (ItemNo:Zone) PRICELIST;

   if %found;
      Price = plPrice;
   else;
      ItemNo = -1;
   endif;

   return;
 /end-free
```

Make sure you add CONST to the code in the /COPY as well.

Oops, I typed ItemNo instead of Price. But, because of CONST this won't compile!

CONST also helps make it self-documenting. You can see which are input and which are output, since the input-only parameters have CONST.

CONST Convienience (1/2)

When the compiler knows that a parameter is input-only, it’s able to do some extra work for you.

```
D GetPrice        PR                  ExtPgm(‘GETPRICE’)
D   ItemNo                       5P 0 const
D   Zone                         1A   const
D   Price                        9P 2

D TempItem        s              5P 0
D TempZone        s              1A
D myPrice         s              9P 2
```

Without CONST:

```
TempItem = 1234;
TempZone = ‘A’;
GetPrice( TempItem: TempZone: myPrice );
```

With CONST:

```
GetPrice ( 1234 : ‘A’: myPrice );
```

You can pass a literal value instead of a variable when you use CONST. The compiler will automatically create a temporary variable, store your literal in it, and pass the temporary variable.

CONST Convienience (2/2)

You can even pass an expression. It will be calculated, stored in a temporary variable, and that temporary variable will be passed:

```
D CalcTax         PR                  ExtPgm(‘CALCTAX’)
D   Subtotal                    11P 2 const
D   Region                       3A   const
D   Total                       11P 2

D TempVar         s             11P 2
   .
   .
```

Without CONST:

```
TempVar = TotalCost – Discounts;
CalcTax( TempVar : Region: Total);
```

With CONST:

```
CalcTax( TotalCost - Discounts : Region: Total );
```

Or the output of a BIF or subprocedure:

BIF Example:

```
OpenFile( %trim(Library) + ‘/’ + %trim(File) );
```

Subprocedure Example:

```
LogError( getErrorMsg (errorNo) );
```

What if I don't want a fixed-size?

Occasionally you want to write a program that will work with any size string that RPG supports. For example, what if you want to write a program that'll center text in a string, no matter how long?

```
      D Center          PR                  ExtPgm('CTR001R4')
      D   String                      65535A   options(*varsize)
      D   Length                         15P 5 const
```

```
       /copy prototypes,center
      D Center          PI
      D   String                      65535A   options(*varsize)
      D   Length                         15P 5 const

      D len             s                10I 0
      D trimlen         s                10I 0
      D start           s                10I 0
      D Save            s             65535A   varying

       /free
             len = Length;
             Save = %trim(%subst(String:1:Len));
             trimlen = %len(Save);
             start = len/2 - trimlen/2 + 1;
             %subst(String:1:len) = *blanks;
             %subst(String:start:trimlen) = Save;
             return;
       /end-free
```

`OPTIONS(*VARSIZE)` disables the compiler's check that you've passed a long enough string.

With `options(*VARSIZE)`, it's up to you to ensure that you don't access memory that you aren't allowed to access. So, be extra careful when you use this!

Tip: `ExtPgm` can help when you're stuck with an ugly naming convention!

*Calling *VARSIZE from CL*

As mentioned earlier, you can call programs with PR/PI from older programs or other languages. The prototype is nice to have, but it's not required when making a call.

```
PGM
    DCL VAR(&TEST) TYPE(*CHAR) LEN(80)

    CHGVAR VAR(&TEST) VALUE('CENTER THIS')
    CALL PGM(CTR001R4) PARM(&TEST 80)

    SNDPGMMSG MSGID(CPF9897) MSGF(QCPFMSG) MSGTYPE(*COMP) +
              MSGDTA(&TEST)
ENDPGM
```

Since there aren't prototypes in CL, you have to use the external name.

Since there's no variable declared, CL's literals use the same rules for determining the variable size as the command line does. Numbers are 15,5, characters are 32 long.

Calling *VARSIZE from RPG

Using the prototype makes it easier to read, and lets you use BIFs, expressions and other tools to make the code easier to write and maintain.

```
D Center          PR                  ExtPgm('CTR001R4')
D   String                   65535A   options(*varsize)
D   Length                      15P 5 const
```

```
 /copy prototypes,center

D ErrMsg          s             50A

 /free

      ErrMsg = 'Invalid Account Number';
      center(ErrMsg: %size(ErrMsg));

      exfmt Screen7;
      *inlr = *on;

 /end-free
```

Always use the prototype name when using CALLP.

Because the 2nd parm is CONST, a BIF can be used to calculate the variable size.

What about optional parms?

It's common to use optional parameters in RPG. They're especially useful when functionality needs to be added to a program without breaking backward-compatibility.

What if you start doing business internationally, and need the GETPRICE program to return the prices in different currencies? Existing programs are fine, but new ones might pass a parameter for the currency type.

This is how that was done with *ENTRY PLIST:

```
C     *ENTRY        PLIST
C                   PARM                    ItemNo
C                   PARM                    Zone
C                   PARM                    Price
C                   PARM                    oCurrency

c                   if        %parms >= 4
c                   eval      Currency = oCurrency
c                   else
c                   eval      Currency = 'us'
c                   endif
```

Options(*nopass)

Making a parameter optional in a prototype can be done the same way you did it before, if you use **`options(*nopass)`**

```
 /copy prototypes,getprice

D GetPrice        PI
D   ItemNo                       5P 0 const
D   Zone                         1A   const
D   Price                        9P 2
D   oCurrency                   32A   const options(*nopass)

D Currency        s                   like(oCurrency)

 /free
     if %parms >= 4;
         Currency = oCurrency;
     else;
         Currency = 'us';
     endif;
```

Remember to add this parm in the /COPY member as well!

OPTIONS(*NOPASS) means that the caller doesn't have to add this parm in order to call this program.

***NOPASS parameters must be at the end of the parameter list. Once you've declared one, any parameters after it must also be *NOPASS.**

Tip: You can include more than one "options" value on a parameter by separating them with colons.

`options(*nopass:*varsize`)

Options(*omit)

A parameter can be declared as "omissible" with options(*omit). Strange as it may sound, this doesn't mean that you don't have to pass the parameter! What it means is that you can pass a special value of *OMIT instead of a variable.

```
 /copy prototypes,getprice

D GetPrice        PI
D   ItemNo                       5P 0 const
D   oZone                        1A   const options(*omit)
D   Price                        9P 2
D   oCurrency                   32A   const
D                                     options(*nopass:*omit)

D Currency        s                   like(oCurrency)
D Zone            s                   like(oZone)

 /free
     if %addr(oZone) = *NULL;
         Zone = 'A';
     else;
         Zone = oZone;
     endif;

     if %parms < 4 or %addr(oCurrency)=*NULL;
         Currency = 'US';
     else;
         Currency = oCurrency;
     endif;
```

When a caller passes *OMIT, the address passed for the parameter is set to *NULL.

When both *NOPASS and *OMIT are specified, you must first check for *NOPASS, and only check *OMIT if the parm was passed.

Calling *NOPASS and *OMIT

Calling a program that uses *NOPASS and *OMIT is easy when you use a prototype.

```
/copy prototypes,getprice

/free

    GetPrice( 54321 : ‘B’: myPrice );

    GetPrice( 54321 : *omit: myPrice );

    GetPrice( 54321 : ‘A’: myPrice: ‘Canada’);

    GetPrice( 12345 : *omit: myPrice: ‘UK’);

    GetPrice( 12345 : *omit: myPrice: *omit );
```

Without a prototype, you can’t use *OMIT (unless you’re calling a subprocedure), but you can still use *NOPASS simply by passing fewer parameters.

Options(*RIGHTADJ)

`Options(*RIGHTADJ)` can be used to tell the compiler to right-adjust a CONST parameter value. (Requires V4R4 or later.)

```
D MyProgram       PR                  ExtPgm(‘MYPGM')
D   Parm1                       20A   const options(*RightAdj)
```

```
 /copy prototypes,MyProgram

 /free

      MyProgram(‘Patio Daddio’);
```

```
 /copy prototypes,MyProgram

D MyProgram       PI
D   Parm1                       20A   const options(*RightAdj)

 /free

   . . . Parm1 now contains “        Patio Daddio” . . .
```

Sadly, I haven’t found a practical use for this feature.

Options(*TRIM)

`Options(*TRIM)` can be used to tell the compiler to remove leading and trailing blanks for a CONST parameter value. (Requires V5R3 or later)

```
D JoinName        PR                  ExtPgm('JOINNAME')
D   First                       30A   varying const options(*trim)
D   Last                        30A   varying const options(*trim)
D   WholeName                   50A
```

```
 /copy prototypes,joinname
D Scott           s             20A   inz('  Scott  ')
D Klement         s             20A   inz('  Klement  ')
D Whole           s             50A
 /free
    JoinName(Scott: Klement: Whole);
   // result is: “Klement, Scott                        “
```

```
 /copy prototypes,joinname
D JoinName        PI
D   First                       30A   varying const options(*trim)
D   Last                        30A   varying const options(*trim)
D   WholeName                   50A
 /free
    // It's not necessary to trim blanks, because the
    // compiler has done it for us.
    Wholename = Last + ', ' + First;
    return;
 /end-free
```

Options(*NULLIND)

`Options(*NULLIND)` tells the system that you want to pass null indicators with a database field. (Requires V5R4 or later)

Without *NULLIND, if a null-capable database field is passed, the called program (or procedure) doesn’t know if is set to null or not, and can’t change whether it’s null or not.

```
D SomeProgram    PR                  ExtPgm(‘SOMEPGM')
D   InvDate                      D   options(*nullind)
```

```
 /copy prototypes,SomePgm
D SomeProgram    PI
D   InvDate                      D    options(*nullind)
 /free
    if %nullind(InvDate);
       %nullind(InvDate) = *OFF;
       InvDate = %date();
    else;
       // Already invoiced.
    endif;
    return;
 /end-free
```

Warning: This is how I expect *NULLIND to work, but I haven’t had a chance to test a V5R4 system yet, so I may be wrong!

Prototypes & External Definitions

Q: I prefer to use an externally defined file as a “data dictionary”. How can I use an external field definition on a prototype?

A: Use LIKE to define the fields in the prototype. Put an externally defined data structure into your /COPY member so you have an external definition to reference.

```
 ** Pull in the external definitions for the CUSTMAS file
D CUSTMAS_t     E DS                   ExtName(‘CUSTMAS’)
D                                      qualified
D                                      based(Template_Only)

D GetCustAddr     PR                   ExtPgm(‘CUSTADDR’)
D   CustNo                             like(CUSTMAS_t.custno)
D                                      const
D   CustName                           like(CUSTMAS_t.name)
D   CustAddr                           like(CUSTMAS_t.addr)
D   CustCity                           like(CUSTMAS_t.city)
D   CustState                          like(CUSTMAS_t.state)
D   CustZip                            like(CUSTMAS_t.zipCode)
```

Data Structures (V5R1+)

Q: Can I pass a data structure using a prototype?

A: You can use LIKEDS to pass a data structure in V5R1 or later.

```
D MyData          DS
D   Field1                      10A
D   Field2                       7P 4

D Example         PR                   ExtPgm(‘EXAMPLE’)
D   DataStruct                         likeds(MyData)

 /free
       callp Example(MyData);
```

Inside the EXAMPLE program:

```
 /copy prototypes,example

D Example        PI
D   DataStruct                        likeds(MyData)

 /free
       DataStruct.Field1 = ‘PARM 1 DATA’;
       DataStruct.Field2 = 19.3412;
```

Data Structures (pre-V5R1)

A: If you don’t have V5R1, you have to use LIKE with pointer logic. (sorry!)

```
D MyData          DS
D   Field1                      10A
D   Field2                       7P 4

D Example         PR                  ExtPgm(‘EXAMPLE’)
D   DataStruct                        like(MyData)

 /free
       callp Example(MyData);
```

Inside the EXAMPLE program:

```
 /copy prototypes,example

D Example        PI
D   DataStruct                       like(MyData)

D LocalVersion   DS                  based(p_data)
D   Field1                      10A
D   Field2                       7P 4

 /free
       p_data = %addr(DataStruct);
       Field1 = ‘PARM 1 DATA’;
       Field2 = 19.3412;
```

Multiple Occurrence DS

This also must be done with pointer logic. Make sure you always pass the first occurrence if you want the whole DS to be passed.

```
D MyData          DS                  occurs(10)
D   Field1                      10A
D   Field2                       7P 4

D Example         PR                  ExtPgm(‘EXAMPLE’)
D   DataStruct                        like(MyData)

 /free
       %occur(MyData) = 1;
       callp Example(MyData);
```

Inside the EXAMPLE program:

```
 /copy prototypes,example

D Example        PI
D   DataStruct                       like(MyData)

D LocalVersion   DS                  based(p_data)
D                                    occurs(10)
D   Field1                      10A
D   Field2                       7P 4
 /free
       p_data = %addr(DataStruct);
       for x = 1 to 10;
         %occur(LocalVersion) = x;
         Field1 = ‘PARM 1 DATA’;
         Field2 = 19.3412;
       endfor;
```

Arrays (1 of 2)

To pass an array, simply code a DIM keyword on the prototype definition:

```
D Months          s             15P 2 dim(12)

D LoadSalesMon    PR                  ExtPgm(‘MONSALES’)
D   Data                        15P 2 dim(12)

 /free
      callp LoadSalesMon(Months);
```

Inside the MONSALES program:

```
 /copy prototypes,MonSales
D LoadSalesMon    PI
D   Data                       15P 2 dim(12)

 /free
       for month = 1 to 12;
          chain month MonthSales;
          if %found;
             Data(month) = msTotal;
          else;
             Data(month) = 0;
          endif;
       endfor;
```

Arrays (2 of 2)

You can use **`options(*VARSIZE)`** if you want to write a program that can work with different sizes of arrays:

```
D LoadSflPage     PI
D   CustNo                       4P 0 const
D   PageSize                     2P 0 const
D   OrderNo                      5A   dim(99) options(*varsize)
D   OrdDate                       D   dim(99) options(*varsize)
D   ShipTo                      25A   dim(99) options(*varsize)
D   Total                       11P 2 dim(99) options(*varsize)
 /free
        for x = 1 to PageSize;

            reade (CustNo) ORDERFIL;
            if %eof;
              Leave;
            endif;


            OrderNo(x) = ofOrder;
            OrdDate(x) = ofDate;
            ShipTo(x)  = ofShipDs;
            Total(x)   = ofTotal;
        endfor;
```

Some programs may call this with a 5 element array. Others with a 20 element. Web applications might want to read 80 or 90 at a time.

Prototypes and Subprocedures

Prototypes can also be used to call Java methods and ILE Subprocedures. There are additional keywords that you can use with those.

- OPDESC
  Pass an operational descriptor (prototype-level)
- EXTPROC
  Provide a separate external name for the subprocedure. This also provides the ability to adjust calling conventions for C, CL or Java. (Prototype-level)
- VALUE
  Pass a parameter by VALUE instead of passing it's address (Parameter level)

Return values:
Subprocedures can return a value that can be used in an expression. This is also part of the prototype.

Not Associated with Prototypes

The following are *NOT* prototype keywords, but are commonly confused with them. These are all data types:

- VARYING
  Varying is a data type. You can specify it on a prototype, just as you'd specify packed, zoned or data data types. It does not affect how the prototype works, but rather defines the data type of one of the parameters. (Just as it does when used on a stand alone variable declaration.)
- PROCPTR
  Specifies that a pointer points to a procedure, rather than data. It's a specific type of pointer.
- CLASS
  Specifies which class a Java object reference belongs to. Again, this helps clarify the data type of the object that you must pass as a parameter. It's a data type, not a prototype keyword.

This Presentation

You can download a PDF copy of this presentation from:

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

Thank you!