ILE Concepts

For the Impatient RPG Programmer

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."

What's ILE?

What is the Integrated Language Environment?

- An environment in which code from many languages can be compiled, bound together, and run.
- First introduced for ILE C in V2R3 (Feb 1993)

  A new environment that lets you write small routines and bind them all together to make programs.
- RPG, COBOL and CL join the party in V3R1 (May 1994)

  RPG's syntax is changed at the same time. The new RPG compiler that has ILE functions is based on the 4th specification of the RPG language, aptly named "RPG IV".
- The original style of programs is now called "OPM"

  OPM = Original Program Model.
- Any to Any Procedure Calls

  Any ILE language can call procedures written in any other language. These procedures can be bound together to make a single program.

It's All About the Call

The purpose of ILE is to provide tools to make it easier to call programs and procedures.

It makes you more productive by making it easier for you to write re-usable tools so that you never have to write the same routine twice.

That's pretty much it. That's all ILE does.

(You can go now.)

ILE Key Concepts

ILE is all about writing modular, reusable code.

- Activation Groups

  Group programs together so they can share resources with one another and be deactivated together.

- Subprocedures

  Subroutines with parameters and local variables. Like "programs within programs."

- Modules

  Subprocedures grouped together into an object.

- Programs

  Modules with one entry point that can be run with the CALL command.

- Service Programs

  Modules with many entry points that can be called from ILE programs.

I don't have time to learn that!

Sometimes people who are new to ILE are put off because the terms *sound* like they're complicated.

- **Activation Groups** -- **Loading & Unloading programs together.**
- **Binding Directories** -- **A list, similar in concept to a library list, that's searched when looking for a subprocedure.**
- **Binder Language** -- **A list of subprocedures in a service program that can be called externally.**
- **Static Binding / Bind by Copy / Dynamic Binding / Bind by Reference** -- **Whether a copy of a subprocedure is included in the program that needs it, or not.**

All of these things sound more complicated than they really are! Don't let the terminology put you off. I'll teach you all of this in 1.5 hours!

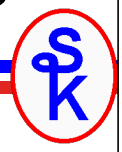
Smaller Pieces Work Better

```
dcl-f PRICELIST disk keyed usage(*input);

// *ENTRY PLIST
dcl-pi *n;
  ItemNo like(plItem) const;
  Zone   like(plZone) const;
  Price  like(plPrice);
end-pi;

chain (ItemNo: Zone) PRICELIST;
if %found;
  Price = plPrice;
else;
  Price = -1;
endif;

return;
```

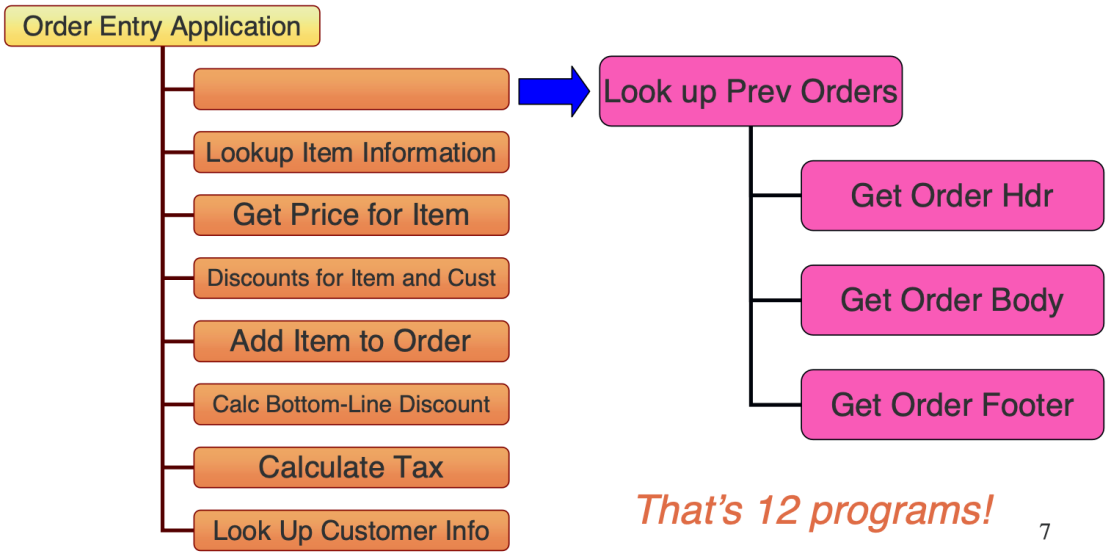
Why are small routines better?

- Takes less time to understand.
- Easier to test, debug, and bullet-proof when it's small.
- Once bullet proofed, it's a "black box" that can be reused from all over.
- If one routine is re-used everywhere, there's only one place to find errors or make changes.

Activation of Many Programs

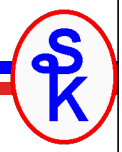


When an application consists of many different sub-programs, each one has to be loaded into memory, and have all of it's files opened, it's variables initialized, etc. Consider a sample application:



That's 12 programs!

Activation Problems



To save time on subsequent calls to each program, you'd leave *INLR turned off. This means:

- The files for all of the programs stay open.
- Record locks can get left behind by mistake.
- Variables remain in their previous state.
- An override will affect all programs, often by mistake.

To unload the programs, what can you do?

- The FREE op-code (ends pgm, but doesn't close files or unlock data areas)
- Call each program with a parm to tell it to turn *INLR = *ON
- RCLRSC (Can close more than you intended!)
- Just leave everything in memory until the job ends.

None of the solutions helps with override problems!

Other languages don't have LR and can't turn LR off!

Taking Out the Trash

Unloading programs is like taking out the trash. When you're ready to throw something away, how do you do it?

- Do you carry each piece of refuse out to a dumpster? (This is what it's like when you call each program to turn on LR.)
- Do you wait until garbage day, have the truck pull into the house, and throw everything into the truck? (That's what RCLRSC is like.)
- Do you wait until you're done with everything in the house, then throw the house away? (That's what SIGNOFF is like.)

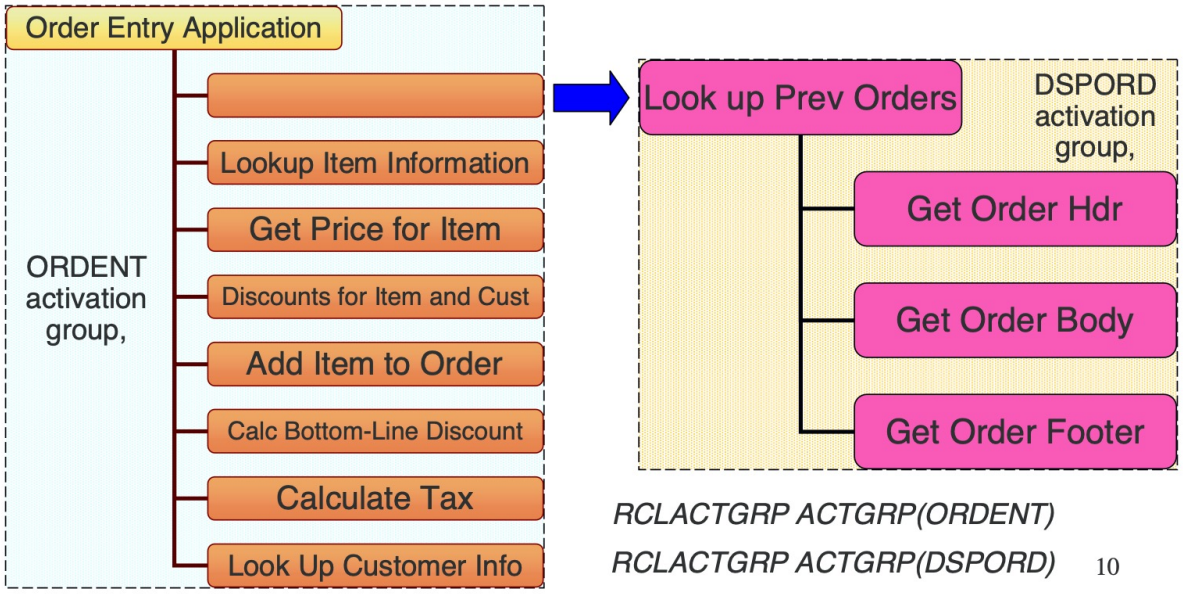
No, of course not. You throw everything into a garbage bag. Then you can discard the whole bag.

- Activation groups are like garbage bags.
- Load your programs into activation groups.
- When you're done, throw away everything in the activation group.

Activation groups are like sub-sections of a job. Maybe think of them as "jobs within a job".

Two Garbage Bags

Putting each application in it's own ACTGRP makes it possible to unload all of it's components at once, without affecting other programs that you may want to leave loaded.



RCLACTGRP ACTGRP(ORDENT)

RCLACTGRP ACTGRP(DSPORD)

Using Activation Groups

Each program or service program is assigned an activation group when you create it. You assign it with the ACTGRP parameter.

- CRTBNDRPG PGM(OEMAIN) ACTGRP(ORDENT)
- CRTBNDRPG PGM(OEITEM) ACTGRP(*CALLER)
- CRTBNDRPG PGM(OECUST) ACTGRP(*CALLER)
- etc.
- CRTBNDRPG PGM(DSPORDMAIN) ACTGRP(DSPORD)
- CRTBNDRPG PGM(ORDHDR) ACTGRP(*CALLER)
- CRTBNDRPG PGM(ORDBODY) ACTGRP(*CALLER)
- etc.

Better yet, you can assign the ACTGRP value in your CTL-OPT (or H-spec), so that you won't forget what to do next time.

```
CTL-OPT DFTACTGRP(*NO) ACTGRP(*CALLER);
```


Special ACTGRP Values

- DFTACTGRP(*YES)
  This means "act like an OPM program". Program is unable to use ILE features. Program is unloaded from memory if LR is on.
- DFTACTGRP(*NO)
  This means "act like an ILE program." Program remains in memory til ACTGRP is destroyed. (Even in default!) LR still closes files and causes variables to be reinitialized on the next call.
- ACTGRP(*CALLER)
  This means "use the same ACTGRP as the program that called me". If called from ACTGRP(ORDENT), this program will run in ORDENT. If called from the command line or an OPM program, it'll run in the default activation group.
- ACTGRP(*NEW)
  Create a new activation group, with a system-generated name, every time this program is called. Automatically destroy that activation group when this program ends.
- ACTGRP(*anything else*)
  Ordinary named activation group. RCLACTGRP must be used to destroy it (or SIGNOFF). *(There's nothing special about QILE!)*

Overrides and Opens

You can scope overrides and shared opens to the activation group so that they won't affect programs in other activation groups.

```
OVRDBF FILE(CUSTMAS) SHARE(*YES) OVRSCOPE(*ACTGRPDFN)
OPNQRYF FILE(CUSTMAS) OPNSCOPE(*ACTGRPDFN)
```

*ACTGRPDFN means:

- **From the default activation group, act like *CALLLVL**
- **From an ILE activation group, only affect that activation group.**

This way, you can control which programs are, and which programs are not, affected by your overrides!

Remember: Activation groups are part of a job. An override scoped to an activation group will not affect other jobs on the system, even if they have activation groups with the same name.

ACTGRPs and Performance

The special value of ACTGRP(*NEW) has received a bad reputation for performance. Part of the reason for this is that people don't understand what's happening:

- It's creating and destroying the activation group that takes the time.
- You can do the same exact thing with ACTGRP(name) and RCLACTGRP and they perform the same (actually, *NEW is slightly faster!)

	200 (CISC) V3R2	*270 (RISC) V4R5*
One time	0.0981 seconds	0.0106 seconds
On a 1,000,000 record file	approx 26 hrs	approx 2.8 hours

Creating an ACTGRP requires work. The CPU has to do something, so it does take time, it's true. *But it's not a problem unless you do it in a loop!*

Main and Sub- Procedures (1/2)

Programs are made up of one or more modules.

Modules consist of one or more procedure.

There are two types of procedures, main procedures and subprocedures.

Main procedures:

- Are what you would normally think of as your “program”
- Is where the RPG cycle runs.
- Are what gets called when your program is called by the CALL command, the CALL op-code, or an EXTPGM prototype.
- Can also be called with a bound call (CALLB or prototype)

Subprocedures:

- Are like subroutines, but with parameters.
- Can have their own local variables.
- Before 6.1 -- Never have F-specs, must use the “global” files from the main procedure.
- Can be called using CALLB or a prototype (without EXTPGM)
- Start and end with P-specs.

Main and Sub- Procedures (2/2)

```
Dcl-F CUSTMAS Usage(*Input) Keyed;

Dcl-PR SUBPROC;
  NoCust like(CustNo);
End-PR;

READ CUSTMAS;
DOW NOT %EOF(CUSTMAS);
  SUBPROC(CustNo);
  READ CUSTMAS;
ENDDO;

*INLR = *ON;

Dcl-Proc SUBPROC;
  Dcl-PI SUBPROC;
    NoCust like(CustNo);
  End-PI;
  COUNT = COUNT + 1;
End-Proc;
```

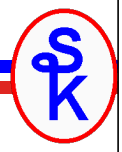
MAIN PROCEDURE

SUB PROCEDURE

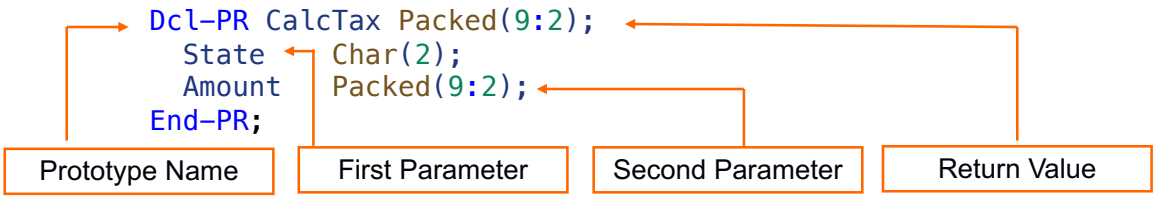
Note:

- Without ExtPgm, prototypes are assumed to call a procedure.
- If you want to refer to a subprocedure by an alternate name, you can use the ExtProc keyword on the PR line.
- Subprocedures are like little programs within a program
- Or maybe like subroutines with their own parameters and variables.

Prototypes

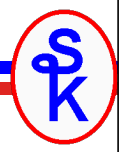


A prototype is very much like a parameter list (PLIST), but is newer and has a lot of additional features. You can use a prototype to call a program, a subprocedure, or a Java class.



- **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 procedure that it calls.
- **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.
- **Return Value (optional)**

Special Keywords

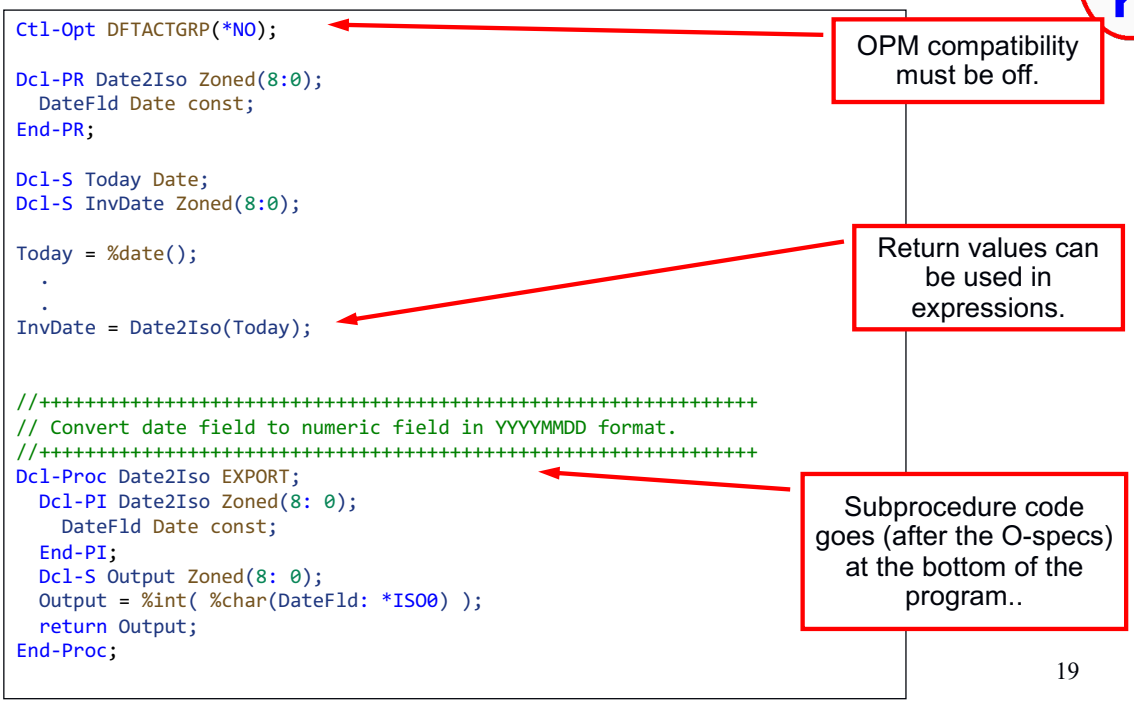


Although subprocedures can be used to call programs and Java methods, there are some prototype keywords that are only used by subprocedures. They are as follows:

- 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) Parameters passed by value do not share memory with their callers, and therefore are "one-way" parameters.

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

Subprocedure Example



```
Ctl-Opt DFTACTGRP(*NO);

Dcl-PR Date2Iso Zoned(8:0);
  DateFld Date const;
End-PR;

Dcl-S Today Date;
Dcl-S InvDate Zoned(8:0);

Today = %date();
  .
  .
InvDate = Date2Iso(Today);


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// Convert date field to numeric field in YYYYMMDD format.
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dcl-Proc Date2Iso EXPORT;
  Dcl-PI Date2Iso Zoned(8: 0);
    DateFld Date const;
  End-PI;
  Dcl-S Output Zoned(8: 0);
  Output = %int( %char(DateFld: *ISO0) );
  return Output;
End-Proc;
```


Modules (1 of 2)

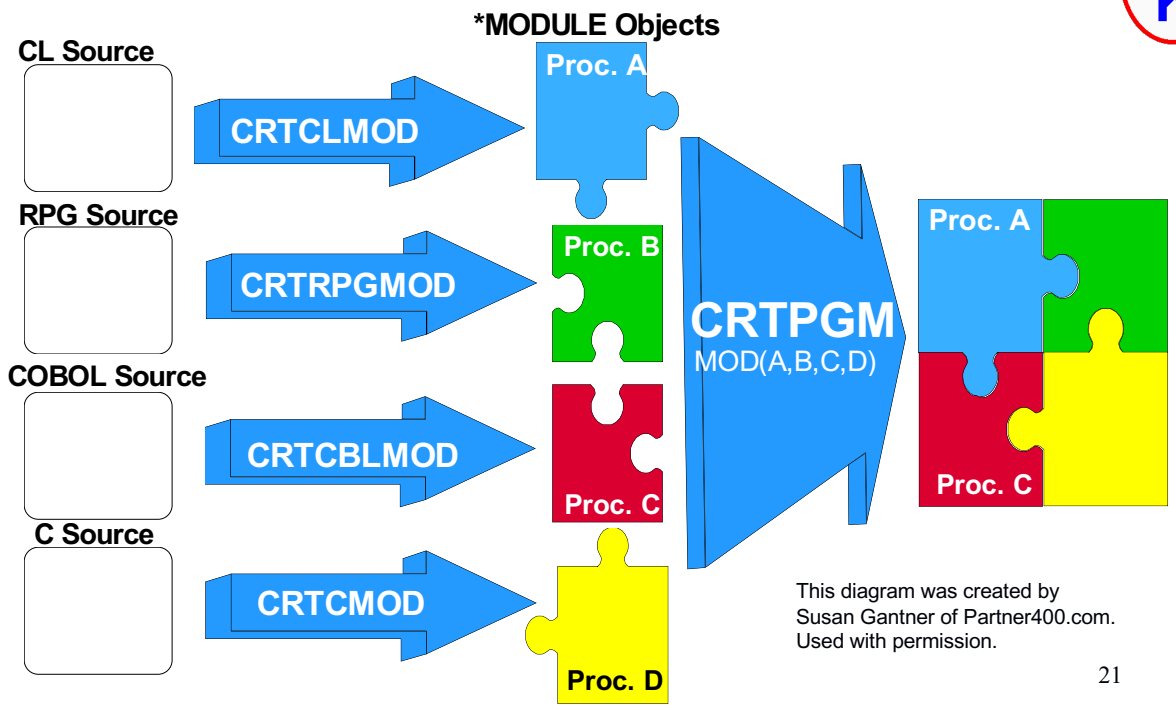
What's a module?

- A source member is compiled into a *MODULE object.
- A module can contain many different subprocedures. Each one can be "exported" so that each one can be called individually.
- A *MODULE object contains compiled and translated code, but that code cannot be run.
- The only use for a *MODULE object is to "bind it" to create a program or service program.
- A program (or service program) requires at least one module, and can consist of many.
- Modules are no longer needed once the program has been created. (However, you might want to save them to speed up future program builds.)

Typically:

- The first module in a program has a main procedure. That main procedure is what's called when the program is first run.
- The rest of the modules have no main procedure, just subprocedures.

Modules (2 of 2)



This diagram was created by Susan Gantner of Partner400.com. Used with permission.

Exporting Subprocedures

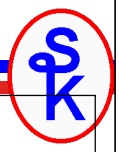
```
Dcl-Proc Date2Iso EXPORT;
  .
  .
End-Proc;
```

Allow subprocs in different modules to call this one..

This Enables:

- You can write a whole library of useful and handy subprocedures.
- Put them all in a module.
- Any subprocedure you want to be called externally can be marked with EXPORT
- Prototypes for these external subprocedures should be put together in a /COPY file.
- Subprocedures that are only used internally are not exported, and their prototypes are not in the /copy file, but instead aren't prototyped, or are placed in the DCL specs of the module.

Sample Module (1 of 6)



Over the next several slides, I'll show you a sample of a typical "NOMAIN" module that contains business logic. It consists of:

- The module code.
- The prototypes member (also has DS and CONST)

Since the prototype member is mostly the same as the PI's that I show in the code, I won't print that member on these slides.

```
**free

Ctl-Opt NOMAIN;

Dcl-F CUSTFILE Usage(*Input) Keyed USROPN;
Dcl-F ORDBYCUST Usage(*Input) Keyed USROPN;

/copy prototypes,custr4

Dcl-PR CEE4RAGE;
  procedure Pointer(*PROC) const;
  feedback Char(12) options(*omit);
End-PR;

Dcl-PR SetError;
  ErrNo Int(10) value;
  Msg Varchar(80) const;
End-PR;

Dcl-S Initialized Ind inz(*OFF);
Dcl-S LastCust Char(8);
Dcl-S save_Errno Int(10) inz(0);
Dcl-S save_ErrMsg Varchar(80) inz('No Error');
```


Sample Module (2 of 6)

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// cust_init(): Initialize customer module
// Note: If you don't call this manually, it'll be called
// automatically when you call another subprocedure
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dcl-Proc cust_Init export;
  if (Initialized);
    return;
  endif;

  open CUSTFILE;
  open ORDBYCUST;

  CEE4RAGE( %paddr(Cust_Done): *omit );
  Initialized = *on;

End-Proc;


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// cust_Done(): Done with module
// Note: If you don't call this manually, it'll be called
// automatically when the activation group is reclaimed
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dcl-Proc cust_Done export;

  if %open(CUSTFILE);
    close CUSTFILE;
  endif;
  if %open(ORDBYCUST);
    close ORDBYCUST;
  endif;
  Initialized = *OFF;

End-Proc;
```

Since there's no RPG cycle, we need to open and close files manually.

You could certainly put other "first run" code here as well.

These don't have to be exported routines, but I like to export them just in case the caller should want to control when our files are opened and closed.

It's expected that most callers won't use these routines.

Sample Module (3 of 6)

```
Dcl-Proc cust_getAddr export;

  Dcl-PI cust_getAddr Ind;
    CustNo Char(8) const;
    Addr likeds(Cust_address_t);
  End-PI;
  Dcl-S Err Int(10);

  Cust_Init();
  chain(e) CustNo CUSTFILE;
  if %error;
    err = %status();
    SetError(CUST_ECHNERR: 'RPG status ' + %char(err)
                         + ' on CHAIN operation.');
    return *OFF;
  endif;

  if not %found;
    SetError(CUST_ENOTFND: 'Customer Not Found');
    return *OFF;
  endif;

  Addr.Name    = Name;
  Addr.Street  = Street;
  Addr.City    = City;
  Addr.State   = State;
  Addr.ZipCode = ZipCode;
  return *ON;

End-Proc;
```

```
Dcl-DS CUST_Address_t qualified based(TEMPLATE);
  Name    Char(25);
  Street  Char(30);
  City    Char(15);
  State   Char(2);
  ZipCode Zoned(9:0);
End-DS;
```

This routine gets a customer's address for the caller.

The DS is defined in the /copy member.

Because the DS in the copy member, the caller can use LIKEDS for their copy, too!

Sample Module (4 of 6)

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// cust_StartOrdList(): Start a list of orders for a customer
//
// CustNo = (input) Customer to get orders for
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dcl-Proc cust_StartOrdList Export;

  Dcl-PI cust_StartOrdList Ind;
    CustNo Char(8) const;
  End-PI;

  Cust_Init();

  setll CustNo ORDBYCUST;
  if not %equal;
    SetError(CUST_ENOORDS: 'No Orders Found for Cust '
             + CustNo );
    return *OFF;
  endif;

  LastCust = CustNo;
  return *ON;
End-Proc;
```

This slide and the next show routines that you might use when reading a list of orders for a customer.

To start a list of orders, just SETLL to get the file positioned in the right place.

Sample Module (5 of 6)

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// cust_ReadOrdList(): Get next order from order list
//
// Ord = (output) Order number of next order
//
// Returns *ON if successful, or *OFF at the end of the list
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dcl-Proc cust_ReadOrdList Export;

  Dcl-PI cust_ReadOrdList Ind;
    Ord Char(5);
  End-PI;

  reade LastCust ORDBYCUST;
  if %eof;
    return *OFF;
  endif;
  Ord = OrderNo;
  return *ON;

End-Proc;
```

This'll be called in a loop. It reads the next order for a customer, and returns the order number.

Why put this in a separate module?

- I may want to use it from 100 places.
- Next year, we may use SQL instead.
- Or maybe a stream file.
- 5 years from now, it might use a web service
- If I find a bug, there's only one place to fix it.
- I can change the underlying file access without changing the callers.

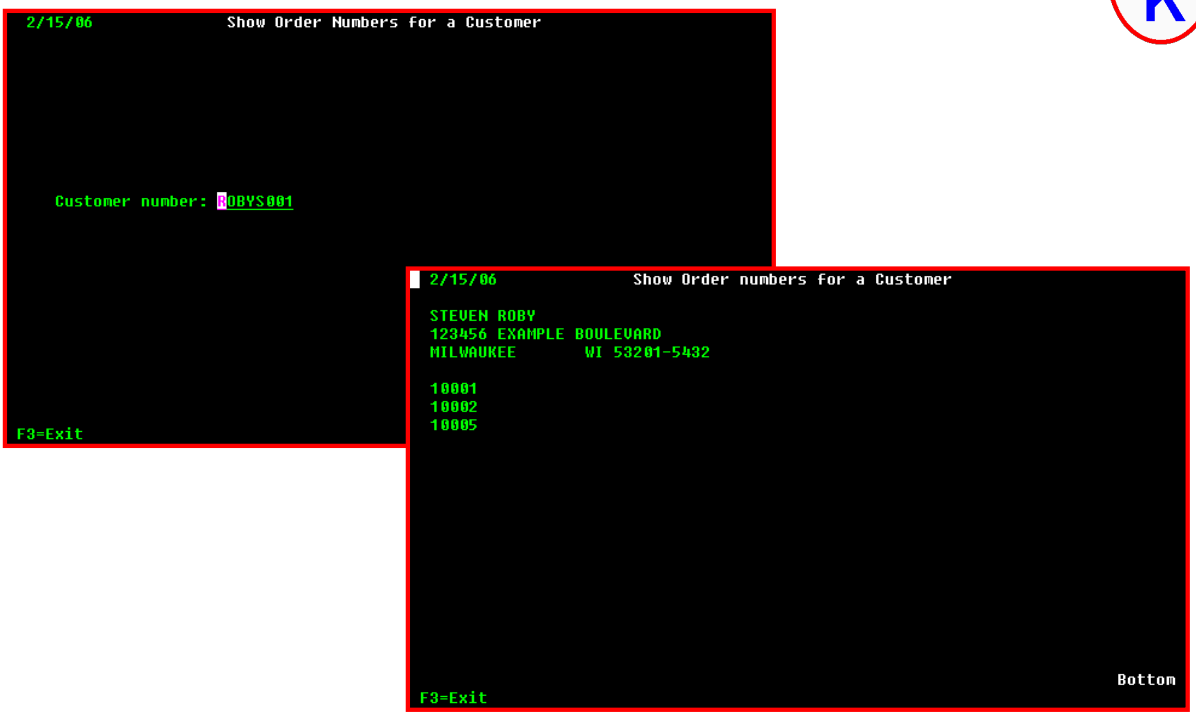
Sample Module (6 of 6)

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// cust_Error(): Get last error that occurred in this module
//
// ErrNo = (output/optional) Error number
//
// Returns the last error message
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dcl-Proc cust_Error Export;
  Dcl-PI cust_Error Varchar(80);
    ErrNo Int(10) options(*nopass:*omit);
  End-PI;

  Cust_Init();

  if %parms>=1 and %addr(Errno)<>*NULL;
    ErrNo = save_Errno;
  endif;

  return save_ErrMsg;
End-Proc;

//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// SetError(): (INTERNAL) set the error number and message
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dcl-Proc SetError;
  Dcl-PI SetError;
    ErrNo Int(10) value;
    Msg Varchar(80) const;
  End-PI;

  save_Errno = Errno;
  save_ErrMsg = Msg;
End-Proc;
```

This is how we're able to communicate error information to the caller.

Error numbers are defined as constants in the /COPY member so that our callers can use them, too!

Calling Sample Module (1 of 3)



Calling Sample Module (2 of 3)

```
**free
Dcl-F SHOWCUSTS WORKSTN SFILE(SFL2: RRN) INDDS(ScreenInds);

/copy prototypes,custr4

Dcl-DS ScreenInds;
  Exit Ind overlay(ScreenInds:03);
  ClearSfl Ind overlay(ScreenInds:50);
  ShowSfl Ind overlay(ScreenInds:51);
End-DS;

Dcl-S  RRN    Packed(4:0);
Dcl-S  Repeat Ind inz(*ON);
Dcl-DS Addr   likeds(Cust_address_t);

dow Repeat;
  exfmt screen1;
  scErrMsg = *blanks;
  Repeat = *Off;

  if Exit;
    *inlr = *on;
    return;
  endif;

  if cust_getAddr(scCust: Addr) = *OFF;
    Repeat = *ON;
    scErrMsg = cust_error();
  endif;
enddo;
```

The first screen asks for a customer number. We'll use it to load the address.

If an error occurs, cust_error() is called to get an error message.

Calling Sample Module (3 of 3)

```
scName   = Addr.Name;
scStreet = Addr.Street;
scCity   = Addr.City;
scState  = Addr.State;
scZip    = Addr.ZipCode;

ClearSfl = *On;
write SFLCTL2;
ClearSfl = *OFF;
RRN = 0;

ShowSfl = cust_StartOrdList(scCust);

dow cust_ReadOrdList(scOrderNo);
  RRN = RRN + 1;
  write SFL2;
enddo;

write SFLFTR2;
exfmt SFLCTL2;

*inlr = *on;
```

The address is loaded into the header record, and the order numbers are loaded into the subfile.

To compile, create the display file, and the two modules.

Then bind the two modules into one *PGM.

Note: CRTxxxMOD is opt 15 from PDM.

```
CRTRPGMOD MODULE(CUSTR4) SRCFILE(mylib/QRPGLESRC)

CRTDSPF FILE(SHOWCUSTS) SRCFILE(mylib/QDDSSRC)
CRTRPGMOD MODULE(SHOWCUST) SRCFILE(mylib/QRPGLESRC)

CRTPGM PGM(SHOWCUST) MODULE(SHOWCUST CUSTR4) ACTGRP(TEST)
```

More About Binding.

Remember, the *MODULE object is only useful for creating programs.

Once the CRTPGM command is done, the *MODULE objects can be deleted, the program will still work. This is because a COPY of all of the compiled code has been included into your program.

This is called "Bind By Copy". (One type of "static" binding.)

Unfortunately, this means that if you used the CUST routines in 50 programs, if you wanted to make a change, you'd have to:

- Re-compile the module.
- Determine which programs use it.
- Re-bind all of the programs.

Fortunately, there's another way. You can bind the module to a special object called a service program (*SRVPGM). Then, you can run the procedures in the service program instead of copying them into your programs.

This is called "Bind By Reference". (Another type of "static" binding.)

Service Programs

A *SRVPGM are very much like regular a regular *PGM. It's an executable object on the system. It contains procedures that you can run. Except:

- Instead of one entry point, a service program has many. (One for each subprocedure.)
- Calls to it can be made from other code (not cmd line).
- You cannot call it with the CALL command. Instead, you call the procedures in it, the same way you'd call any other subprocedure.
- Calls to a service program (or a bound module) are much faster than dynamic (traditional "CALL command") calls.

Binder Language (1 of 3)

Since a service program contains many subprocedures, you have to tell it which ones can be called externally. This is done using "Binder Language".

Don't let the word "language" fool you. Binder language is very simple, it's only job is to list the procedures you want to export.

```
STRPGMEXP SIGNATURE('CUSTR4 Sig Level 1.00')
   export symbol(cust_init)
   export symbol(cust_done)
   export symbol(cust_getAddr)
   export symbol(cust_StartOrdList)
   export symbol(cust_ReadOrdList)
   export symbol(cust_Error)
ENDPGMEXP
```

The SIGNATURE parameter works like level checks do. When you bind a program, it remembers the signature. If the signature changes, you'll get a "Signature Violation" error.

Binder Language (2 of 3)

A program calls subprocedures in a service program by number.

```
STRPGMEXP SIGNATURE('CUSTR4 Sig Level 1.00')
  export symbol(cust_init)              -- 1
  export symbol(cust_done)              -- 2
  export symbol(cust_getAddr)           -- 3
  export symbol(cust_StartOrdList)      -- 4
  export symbol(cust_ReadOrdList)       -- 5
  export symbol(cust_Error)             -- 6
ENDPGMEXP
```

If you rearrange the procedures in the binder source, a program could end up calling the wrong one.

1. If your program was set up to cust_getAddr(), it'll remember it as #3.
2. If you later recompile the service program, but add a new subprocedure at the top of the list, cust_done() might become #3!
3. The program would then call cust_done() when it was supposed to call cust_getAddr()!

The moral of the story? Always add new procedures to the end so that the existing numbers won't change. Then you don't have to re-bind the callers!

If you absolutely must change the order of the procs, change the signature to protect you from calling the wrong procedures.

Binder Language (3 of 3)

```
STRPGMEXP SIGNATURE(*GEN) PGMLVL(*CURRENT)
  export symbol(cust_init)
  export symbol(cust_done)
  export symbol(cust_getAddr)
  export symbol(cust_StartOrdList)
  export symbol(cust_ReadOrdList)
  export symbol(cust_Error)
ENDPGMEXP

STRPGMEXP SIGNATURE(*GEN) PGMLVL(*PRV)
  export symbol(cust_init)
  export symbol(cust_done)
  export symbol(cust_getAddr)
ENDPGMEXP
```

It's possible to have more than one signature block, and have the OS generate the signatures based on the names and order of the exports.

However, I've never found any benefit to doing so. After 5 or 10 changes, it becomes very awkward to keep adding more signature blocks!

Building Service Programs

You build service programs from *MODULE objects, just like programs. The only difference is that you use CRTSRVPGM instead of CRTPGM.

```
CRTRPGMOD MODULE(CUSTR4) SRCFILE(mylib/QRPGLESRC)

CRTSRVPGM SRVPGM(CUSTR4) MODULE(CUSTR4) EXPORT(*SRCFILE)
          SRCFILE(mylib/QSRVSRC) ACTGRP(*CALLER)

CRTDSPF FILE(SHOWCUSTS) SRCFILE(mylib/QDDSSRC)
CRTRPGMOD MODULE(SHOWCUST) SRCFILE(mylib/QRPGLESRC)

CRTPGM PGM(SHOWCUST) MODULE(SHOWCUST) BNDSRVPGM(CUSTR4)
```

Note the following:

- The SRCFILE listed on the CRTSRVPGM command is the binder source and *not* the RPG source.
- Changing from a *MODULE bound by copy to a *SRVPGM bound by reference only required creating a list of exports, and changing the commands used to compile. *(I didn't change my RPG code at all!)*

Binding Directories

Binding directories are similar to library lists. They contain a list of *SRVPGM and *MODULE objects.

- Instead of listing every object on on the CRTPGM command, use a BNDDIR
- The binder will search the BNDDIR for a needed subprocedure.
- The binder will only bind your pgm/srvpgm to objects containing procedures that you call. The others won't be bound to your program.

```
CRTBNDDIR BNDDIR(QGPL/MYSRVPGMS)
ADDBNDDIRE BNDDIR(QGPL/MYSRVPGMS) OBJ(*LIBL/CUSTR4 *SRVPGM)
ADDBNDDIRE BNDDIR(QGPL/MYSRVPGMS) OBJ(*LIBL/ORDERS *SRVPGM)
ADDBNDDIRE BNDDIR(QGPL/MYSRVPGMS) OBJ(*LIBL/UTILS *SRVPGM)
ADDBNDDIRE BNDDIR(QGPL/MYSRVPGMS) OBJ(*LIBL/ACCTRCV *SRVPGM)
```

For many shops, creating one binding directory for all of their service programs is ideal. Then the only special step needed to compile programs (and get all of the routines they need) is to specify the binding directory:

```
CRTPGM PGM(SHOWCUST) MODULE(SHOWCUST) BNDDIR(MYSRVPGMS)
```

CRTBNDRPG

You can even include a BNDDIR on your H-spec so that you don't have to remember it:

```
H BNDDIR('QGPL/MYSRVPGMS':'OTHERDIR': 'EXAMPLE')
```

The CRTBNDRPG command is a shortcut for CRTRPGMOD (to QTEMP) followed by CRTPGM for a single module.

You can specify a BNDDIR on the CRTBNDRPG command, or your H-spec, to tell CRTBNDRPG to bind to modules or service programs.

If you add the following H-specs to the SHOWCUST example program

```
H DFTACTGRP(*NO) ACTGRP('TEST')
H BNDDIR('QGPL/MYSRVPGMS':'OTHERDIR': 'EXAMPLE')
```

Then you could compile it as follows (very close in simplicity to CRTRPGPGM!)

```
CRTBNDRPG PGM(SHOWCUST) SRCFILE(mylib/QRPGLESRC)
```

… or you can use PDM option 14. (Which will call CRTBNDRPG)

This Presentation

You can download a PDF copy of this presentation from:

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

Thank you!