CODE/400 for Windows

Selected Advanced Topics

Hands On Lab

COMMON

Vadim Berestetsky, Edmund Reinhardt and the AS/400 Team

IBM Canada Ltd

Technical Information and Education

For more technical information on CODE/400 or VisualAge for RPG please visit us at our web site:

http://www.software.ibm.com/ad/varpg

or contact either

Dave Slater at slater@ca.ibm.com
Claus Weiss at weiss@ca.ibm.com

First Edition (March 1999)

Table of Contents

Introduction

The **CoOperative Development Environment/400**, better known as **CODE/400,** is a set of integrated development tools that allow you to: create, edit, compile, and maintain your source code; debug programs using a PC connected to an AS/400; and completely organize your programming projects.

The CODE/400 product includes the following tools:

- **CODE Editor**
  A powerful language-sensitive editor that you can easily customize. Token highlighting of source makes the various program elements stand out. It has SEU- like specification prompts for RPG and DDS to help enter column-sensitive fields. Local syntax checking and semantic verification for your RPG, COBOL and DDS source makes sure it will compile cleanly the first time on an AS/400. If there are verification errors, an Error List lets you locate and resolve problems quickly. On-line programming guides, language references, and context-sensitive help make finding the information you need just a keystroke away.
- **CODE Program Generator**
  An interface that allows you to submit requests to the AS/400 to compile, bind, or build objects on the host. The tool gives you easy access to all the compile options available for all the supported create commands (CRTxxx).
- **CODE Designer**
  A rich graphical interface that makes designing or maintaining display file screens and printer file reports easy and fun.
- **CODE Debugger**
  A source-level debugger that allows you to debug an application running on a host AS/400 from your workstation. It provides an interactive graphical interface that makes it easy to debug and test your host programs.
- **CODE Project Organizer**
  An enhanced and more flexible workstation version of the Program Development Manager (PDM). It ties all the parts of CODE/400 together and allows you to quickly access all the power of CODE/400 and to effectively manage and organize your development projects.

Goal

In this session, you will learn some nontrivial features and functionality of the CODE/400 tools by playing with them. We will learn how to customize the LPEX editor by using predefined functions and extending its capability with REXX macros and Java Lpexlets. You will also find out how productive CODE/400 even when there is no connection to the AS/400 host. We are confident that CODE/400 will save you time and effort in your day-to-day programming tasks. It will make you a more efficient and effective programmer. At the same time, it will save cycles on your AS/400. Now let's spend a couple of hours playing and see if you agree.

Tool

Installing CODE/400 for Windows

The CODE/400 for Windows product consists of two parts:

1. The 'back-end' which resides on the AS/400.
   This part is responsible for handling all the workstation requests such as getting or saving source members, etc. The back-end is shipped with the ADTS host utilities (SEU, PDM, DFU, SDA, ...).
2. The 'front-end' which is installed on your workstation.
   These workstation files can be installed from:
   - a local CD drive
   - a LAN drive (assuming that an installable image has been set up on the LAN)
   - an AS/400 (assuming that the workstation files have been installed into an AS/400 shared folder called QADTSWIN).

   The workstation install uses the Windows industry standard InstallShield program.

The minimum hardware requirements for CODE/400 are a 486 computer with 16MB of memory, and a SVGA monitor. For zippy performance, the recommended workstation hardware is a Pentium computer, with 32MB of memory, and an SVGA monitor. A complete install of CODE/400 with the help for all supported languages uses about 60MB of disk space.

The Lab - Section 1: Customizing the CODE Editor

Section Introduction

Basic Editor Features

The CODE Editor has all the basic functions that you would expect in any serious editor:

- Cut, copy, and paste
- Block marking of lines, characters, or rectangles and with copy, move, overlay, and delete operations.
- Powerful find and replace functionality.
- Unlimited undo and redo.
- Automatic backup and recovery.

In addition there are a few more functions that you may not have seen in a workstation editor:

- Token highlighting -- different language constructs are highlighted using different colors and fonts to help identify them in a program. This highlighting is completely customizable (see the menu item **Options** → **Token attributes**...).
- SEU- like format-line rulers to show the purpose of each column for column-sensitive languages like RPG and DDS. These rulers can automatically update themselves to reflect the current specification.
- SEU-like specification prompting for RPG and DDS.
- Sequence numbers which allow SEU-style commands in the prefix area.
- Intelligent tabbing between columns for column-sensitive languages.
- Automatic uppercasing for languages that expect uppercase.
- For column-sensitive languages there is the new CODE FIELDS ON command that simplifies text insertions and deletions.
- On-line language reference help.

Editor Programming (ultimate customization)

Despite its rich functionality, the CODE editor may still lack features that suit needs of a particular AS/400 shop, or even individual programmers. Therefore, we provide a means of customizing the editor to your liking. You can:

- Specify default editor settings.
- Add editor functions and your own macros to the menus and toolbars.
- Assign/re-assign keys and/or line commands to editor functions and your own macros.
- Interact with the host via the **CODESRV** command.
- Implement and execute REXX macros and Java Lpexlets.

Code/400 - Advanced topics: Hands On Lab

In this section we will introduce you to:

- Associating name patterns with source types.
- Associating source types with language profiles.
- CODE editor commands.
- REXX macros for the CODE/400 editor.
- Adding and updating editor menus and popup menus.
- Updating the editor toolbar.
- The CODESRV command.
- Working with various editor profiles.

You will:

- 1. Associate RPGLE file types with all local files that have a *.RPG extension.
- 2. Learn, execute and master various LPEX editor commands.
- 3. Write and execute the RPGPROC REXX editor macro (that uses prompt box).
- 4. Update the editor menu, popup menu, and toolbar.
- 5. Use the CODESRV to submit remote commands.
- 6. Understand editor profiles, and create an RPGLE400.LXU profile.

Now let’s begin our journey into wonderful world of CODE/400...

Step 1. Connecting to the AS/400

PURPOSE:

Communications between the AS/400 and your workstation can be configured for:

•TCP/IP communications using the native Windows built in TCP/IP support. You can use any 5250 emulator that supports TCP/IP.

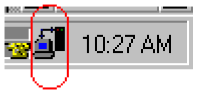
• SNA (System Network Architecture) / APPC (advanced program-to-program communications). This setup requires either: Client Access; Personal Communications; or RUMBA to handle the communications.

For this lab session, you will use TCP/IP communications

INSTRUCTIONS:

1a. From the **Start → Programs → VisualAge RPG and CODE400** menu, select **TCPIP Communications Server**. This starts the CODE Daemon on your workstation. This program waits and listens for an AS/400 to contact it on a specific TCP/IP port and then makes a connection.

An icon will appear in your system tray (bottom right of your screen).

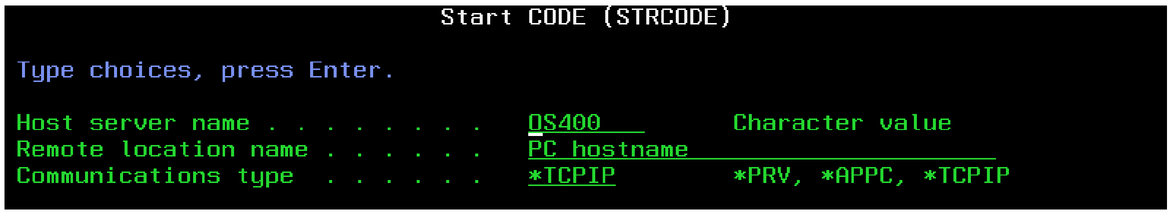


1b. **Start** a 5250-emulation session.

1c. Sign on to the AS/400. Your userid and password should both be **CODELABxx** where **xx** is your workstation number (01, 02, etc.). The **Enter** key could be the **Ctrl** key in your 5250-emulation session.

1d. At the AS/400 command line type: **STRCODETCP**. This will call a CL program which automatically figures out which IP address your emulator is using and invokes the STRCODE command. You should see a screen that has **EVFCLOGO** in the upper left-hand corner.

If you did not have this CL program, at the AS/400 command line type (or prompt) the command: STRCODE RMTLOCNAME(PC_hostname) CMNTYPE(*TCPIP)
You should see a screen that has **EVFCLOGO** in the upper left-hand corner.



```
                           Start CODE (STRCODE)

Type choices, press Enter.

Host server name . . . . . . . .   OS400          Character value
Remote location name . . . . . .   PC hostname
Communications type  . . . . . .   *TCPIP         *PRV, *APPC, *TCPIP
```

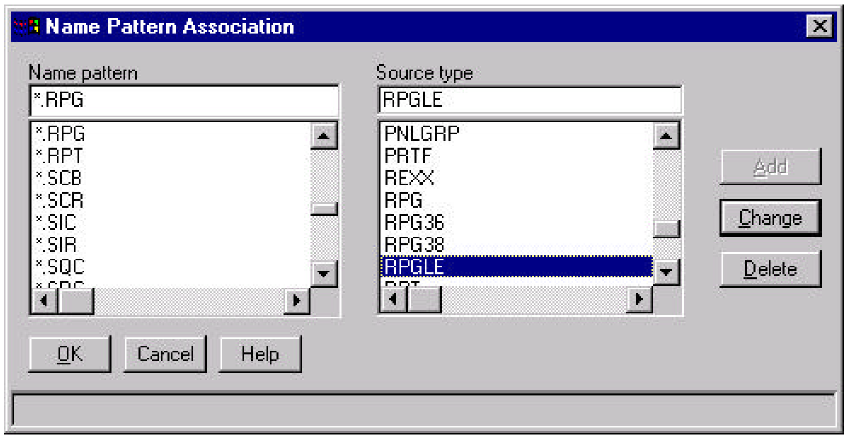
Step 2. Associating name patterns with source types

PURPOSE:

For the following exercises we will need to create an ILE RPG file and store it on the local drive. Most local source files have both a file name and a file extension. The CODE editor uses the file extension to determine what is in the file. For example, files having an .RPG file extension are assumed to contain OPM RPG while files with an .IRP extension are assumed to be ILE RPG. It's easy for us to change these default settings. In the following exercise you will associate the name pattern ***.RPG** with ILE RPG instead of OPM RPG.

INSTRUCTIONS:

2a. Open an MS-DOS window. Type **CD C:\ADTSWIN\EXTRAS** and press **Enter**. Start the CODE/400 editor by typing the **CODEEDIT** command at the MS-DOS prompt.
2b. From the editor '**Options**' menu, select '**Associations**' **->** '**Name patterns**' option. The 'Name Pattern Association' dialog comes up.



2c. From the 'Name pattern' list box pick the ***.RPG** pattern. Select the **RPGLE** value from the 'Source type' list box.
2d. Press the '**Change**' button to make the changes take effect.
2e. Press the '**Ok**' button to dismiss the 'Names Pattern Association' dialog.
From now on when we open a file with a .RPG extension, it will be an **ILE RPG** file.

NOTE: You can associate source types with the name patterns for host files as well. For example, associating a ***/QRPGSRC(*)** pattern with the **RPG** source type tells the editor to treat any member from the QRPGSRC file as an OPM RPG file.

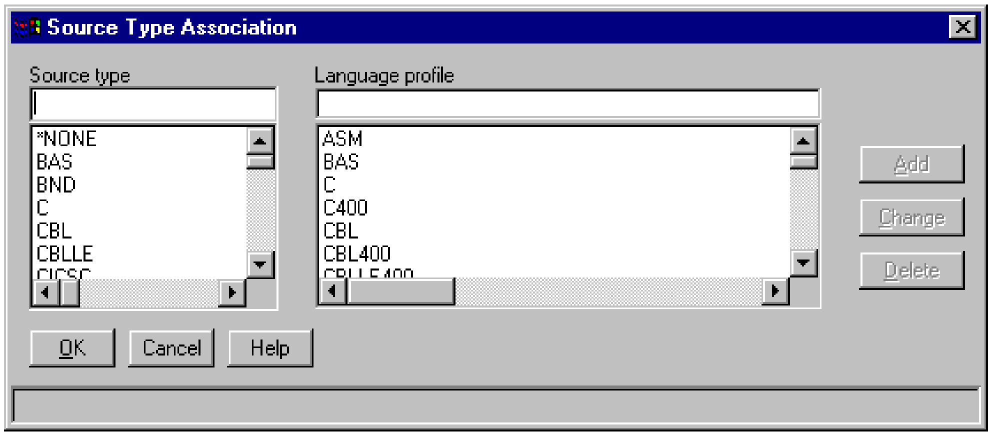
Step 3. Associating source types with language profiles

PURPOSE:

In the following exercise you will see the importance being able to associate name patterns with source types. The CODE editor gives you the flexibility of executing editor commands and macros when the file gets loaded into the editor. Moreover, different commands and macros get executed for different 'language profiles'. Therefore, it is very important that file source types are associated with the appropriate language profiles. Guess what, CODE/400 provides you with such a feature!

INSTRUCTIONS:

3a. From the editor '**Options**' menu, select '**Associations**' -> **'Source types'** option. The 'Source Type Association' dialog comes up

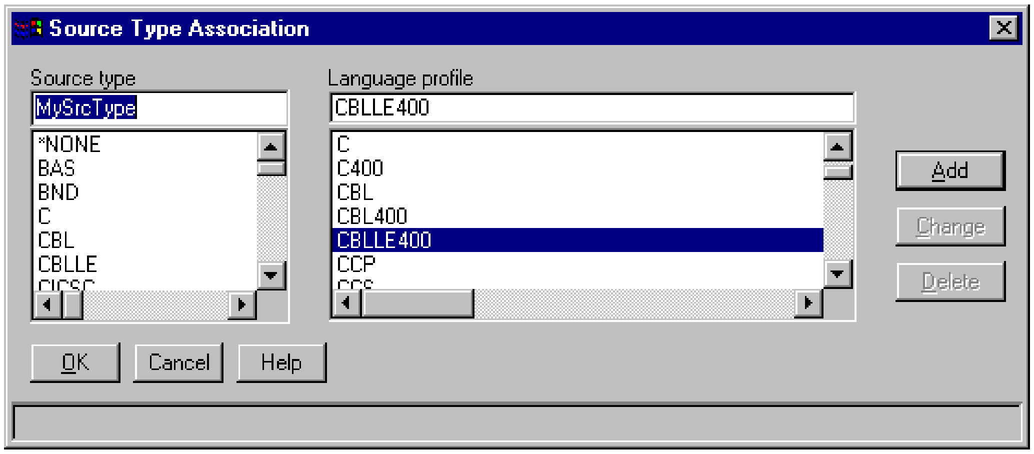


3b. From the 'Source type' list box (on the left) select the **RPGLE** source type. Notice how the **RPGLE400** language profile gets selected in the 'Language profile' list box (on the right).

NOTE: In Step 2 of this section you associated the **RPGLE** source type with the ***.RPG** name pattern. We also just saw that **RPGLE** source type is associated with the **RPGLE400** language profile. This actually means that whenever we open a local file with **.RPG** extension, editor commands and macros in **RPGLE400** language profile get executed!

Now let's get a bit creative. We will invent a new source type called 'MySrcType' and associate it with the CBLLE400 language profile (which stands for ILE COBOL).

<u>**3c.**</u> In the 'Source type' entry field type: **MySrcType** and then select CBLLE400 from the 'Language profile' list box.



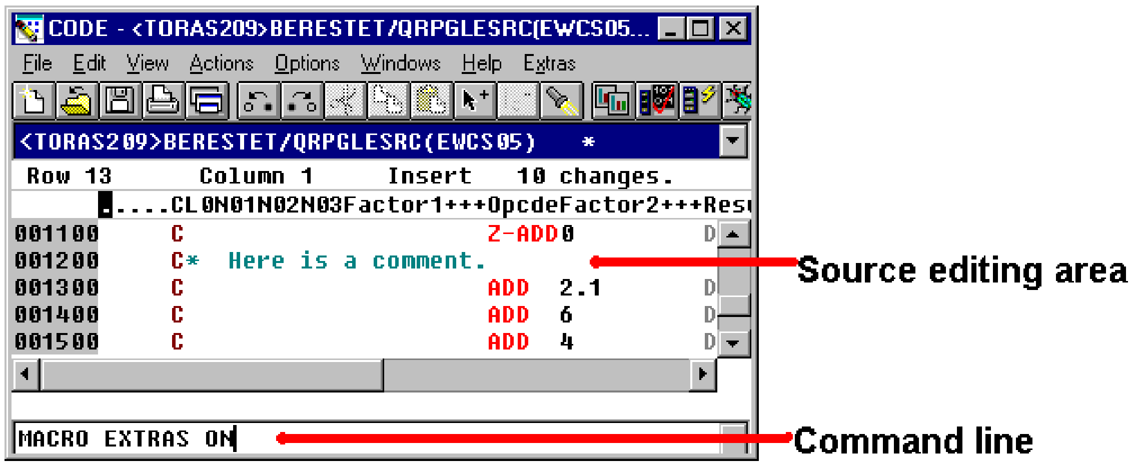
<u>**3d.**</u> Press the 'Add' button to complete the association.

<u>**3e.**</u> Press the 'Ok' button to dismiss the 'Source Type Association' dialog.

Step 4. Executing existing REXX macros

PURPOSE:

To get comfortable with running REXX macros from the CODE/400 editor you will now execute two macros that are currently shipped with the CODE/400 product. In order to execute a REXX macro you have to switch to go to the command line. Press '**ESC**' key to switch between the source editing area and command line:


CODE - <TORAS209>BERESTET/QRPGLESRC(EWCS05...
File Edit View Actions Options Windows Help Extras
Source editing area
Command line

```
<TORAS209>BERESTET/QRPGLESRC(EWCS05)    *
 Row 13        Column 1      Insert    10 changes.
          ....CL0N01N02N03Factor1+++OpcdeFactor2+++Res
001100       C                      Z-ADD0             D
001200       C*   Here is a comment.
001300       C                      ADD    2.1         D
001400       C                      ADD    6           D
001500       C                      ADD    4           D

MACRO EXTRAS ON
```

REXX macros are usually run by typing the following command: **MACRO MacroName**. If you are certain that there is no other editor command that matches the name of your macro then the **MACRO** directive can be omitted.

INSTRUCTIONS:

Part1: Running a simple REXX macro

4aI. Press the **Esc** key to go to the command line.

4bI. Type **MACRO EXTRAS ON** and then press **Enter**. You have just run your first editor macro ! The EXTRAS macro is used to update the path that the editor searches when an editor command or macro is executed. By issuing the command, "EXTRAS ON" the editor will search the ADTSWIN\EXTRAS directory and then the ADTSWIN\MACROS directory. It remains on until it is explicitly turned off (EXTRAS OFF). The EXTRAS directory contains the additional macros that you are about to play with.

__4cI__. Open a file with sequence numbers by typing

LX <OS400>CODELABxx/QRPGLESRC(PAYROLL)

on the editor command line and pressing the **Enter** key. **LX** is the editor command used to a file.

__4dI.__ Enter about 10 lines of text into the file. It doesn't matter what it is.

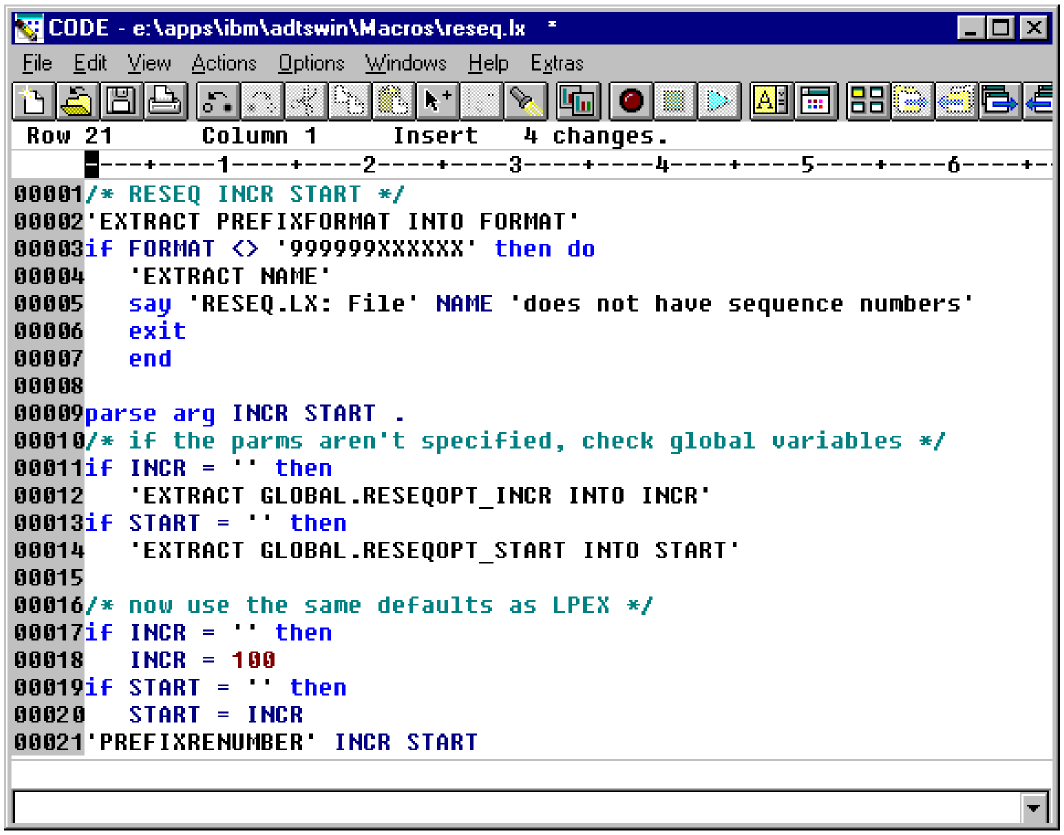
__4eI.__ Go to the fifth line and delete it by pressing **Ctrl+Backspace**.
Notice that the sequence numbers now skip 000010.

__4fI__. On the editor command line type **MACRO RESEQ** and then press **Enter**. This will resequence the file using the values in the **Set Resequence Options** dialog available from the **'Options** ' **->** '**Resequencing**' pull down.
Notice that the fourth line of text that you have entered, now has sequence number 3.

__4gI.__ RESEQ is a macro written in REXX. Type:

LX RESEQ.LX

and then press **Enter** to open the macro to see what it does. It may look a little cryptic now but we will try to resolve the mystery.


CODE - e:\apps\ibm\adtswin\Macros\reseq.lx *
File Edit View Actions Options Windows Help Extras

```
Row 21        Column 1       Insert     4 changes.
/* RESEQ INCR START */
'EXTRACT PREFIXFORMAT INTO FORMAT'
if FORMAT <> '999999XXXXXX' then do
   'EXTRACT NAME'
   say 'RESEQ.LX: File' NAME 'does not have sequence numbers'
   exit
   end

parse arg INCR START .
/* if the parms aren't specified, check global variables */
if INCR = '' then
   'EXTRACT GLOBAL.RESEQOPT_INCR INTO INCR'
if START = '' then
   'EXTRACT GLOBAL.RESEQOPT_START INTO START'

/* now use the same defaults as LPEX */
if INCR = '' then
   INCR = 100
if START = '' then
   START = INCR
'PREFIXRENUMBER' INCR START
```

NOTE: Please **do not** close the PAYROLL file before you get to the next exercise.

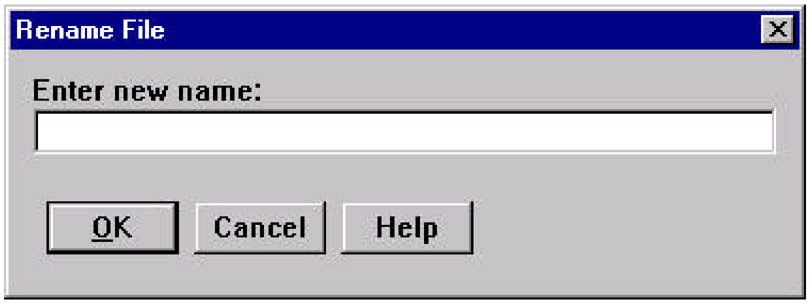
Part2: Running a REXX macro with the prompt

At times it may be required to prompt the user for some information. REXX in conjunction with the CODE editor commands allow for a simple, one-line prompt box, which is good enough for many cases. Let's try an example:

4aII. Notice that EXTRAS is still ON from the previous exercise. You will see a new editor menu called '**Extras**'. Play with the options that are available from that menu. You can get more information about the supplied 'extra features' by exploring the 'Extra Features Guide' available from the '**Extras**' -> '**Information**' menu.

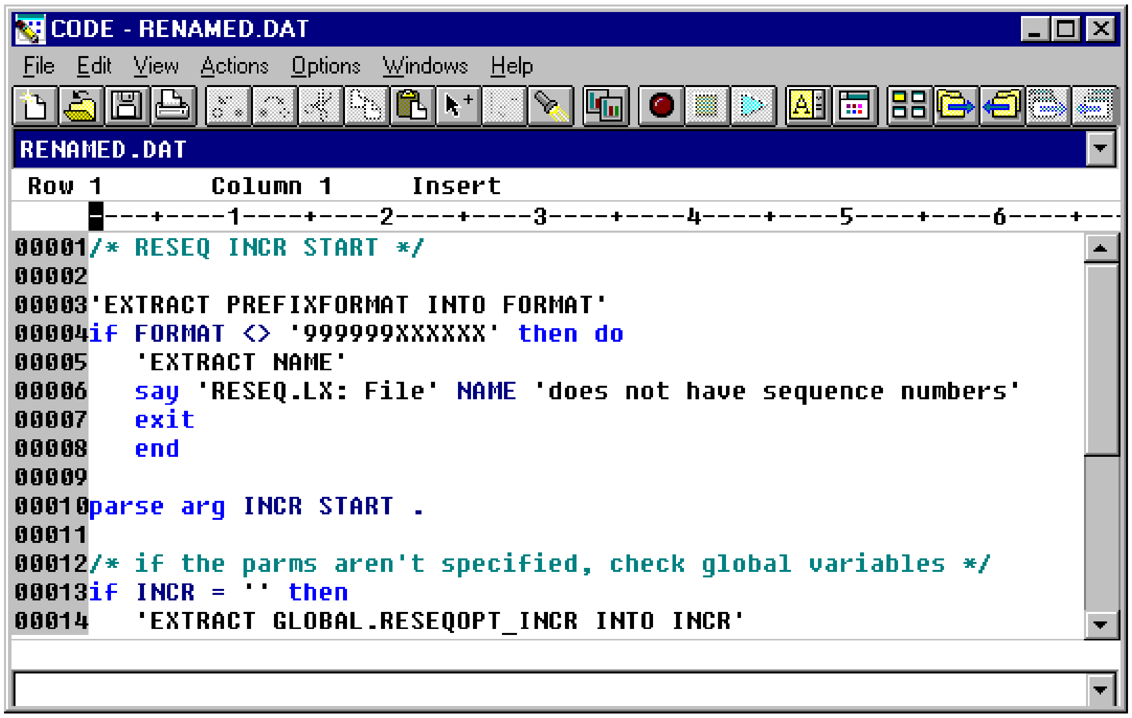
4bII. Press the **Esc** key to go to the command line

4cII. Type **MACRO RENAME** and then press **Enter.** The following dialog box comes up:



4dII. Enter **RENAMED.DAT** in the 'Rename File' entry box for the new file name and then press the '**Ok**' button.

4eII. The 'Rename File' entry box disappears, and the file that is currently loaded in the editor gets saved under its new name - **RENAMED.DAT**



```
/* RESEQ INCR START */

'EXTRACT PREFIXFORMAT INTO FORMAT'
if FORMAT <> '999999XXXXXX' then do
   'EXTRACT NAME'
   say 'RESEQ.LX: File' NAME 'does not have sequence numbers'
   exit
   end

parse arg INCR START .

/* if the parms aren't specified, check global variables */
if INCR = '' then
   'EXTRACT GLOBAL.RESEQOPT_INCR INTO INCR'
```

__4fII__. As you might have suspected already, **RENAME** is another REXX macro. Type:

LX RENAME.LX

and then press the **Enter** key to bring up its source in the editor.

__4gII__. While looking through the source, pay particular attention to the following lines

```
'set lineread.title Rename File'
'set lineread.prompt Enter new name:'
'lineread 255'
```

These lines:

1) Set the dialog title to “Rename file”
2) Create a dialog label called “Enter new file:”
3) Read up to 255 characters from the entry field.

You will use similar code in the following exercises when a need for a prompt dialog box arises.

Step 5. Creating an RPGPROC macro

PURPOSE:

Commenting code is seldom done well. Programmers are usually too busy just trying to write the code and make it work to ever have time to go back and add comments. But leaving out comments makes code maintenance difficult. What if we could somehow automate this process? Let's write a little REXX macro that prompts the user for the procedure name and then generates an appropriate procedure template that includes lovely comments!

INSTRUCTIONS:

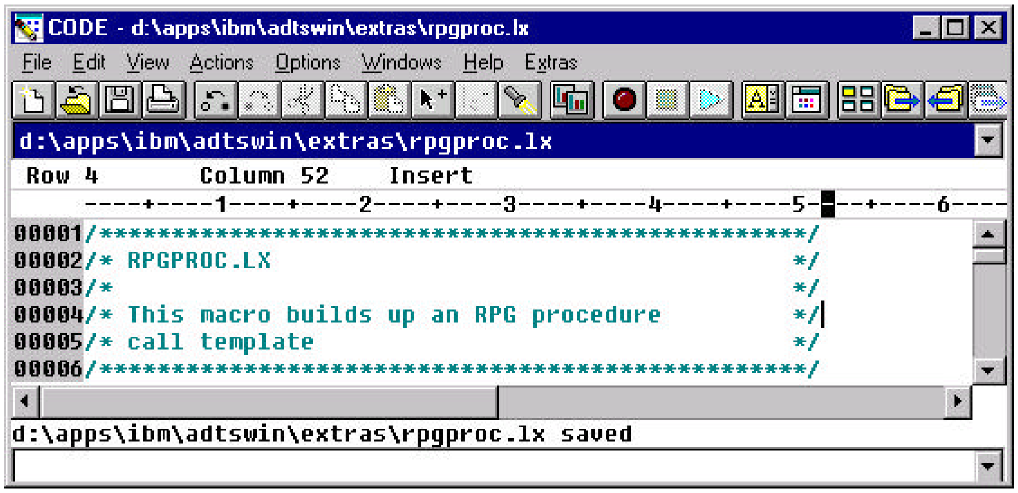
5a. Press the **Esc** key to go to the command line.

5b. Open a new file called RPGPROC.LX by typing

LX RPGPROC.LX

and then press the **Enter** key.

5c. It is necessary to start every REXX program with a comment. The first few lines will give a brief description of what our macro will do. Type them in:


CODE - d:\apps\ibm\adtswin\extras\rpgproc.lx
File Edit View Actions Options Windows Help Extras
d:\apps\ibm\adtswin\extras\rpgproc.lx
Row 4 Column 52 Insert

```
/****************************************************/
/* RPGPROC.LX                                      */
/*                                                 */
/* This macro builds up an RPG procedure           */
/* call template                                   */
/****************************************************/
```

d:\apps\ibm\adtswin\extras\rpgproc.lx saved

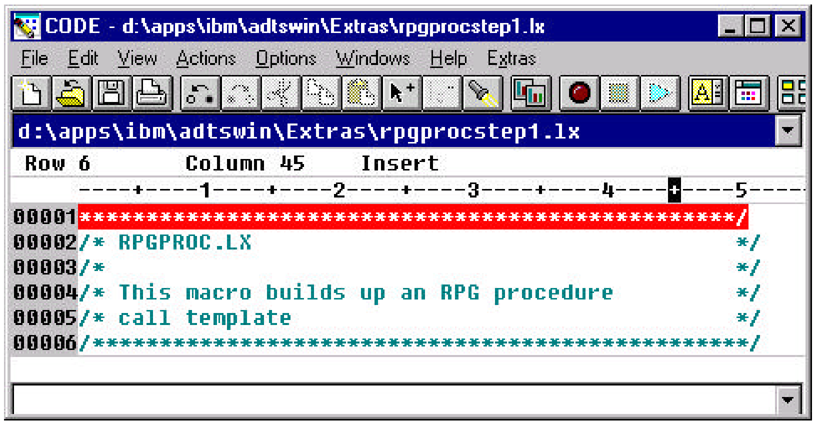
5d. At this point you should save the file. Use the **'File'->'Save'** menu option. Now you can actually run this new macro. Of course, it won't do anything yet because the macro only contains comments.

5e. Switch to the command line (press the **Esc** key) and type **MACRO RPGPROC**. Nothing happens.

5f. Just to get more comfortable with the REXX environment, let's make a syntax error in the REXX program. On the first line remove the first forward slash '/' character, so that the line becomes:

```
***************************************************/
```

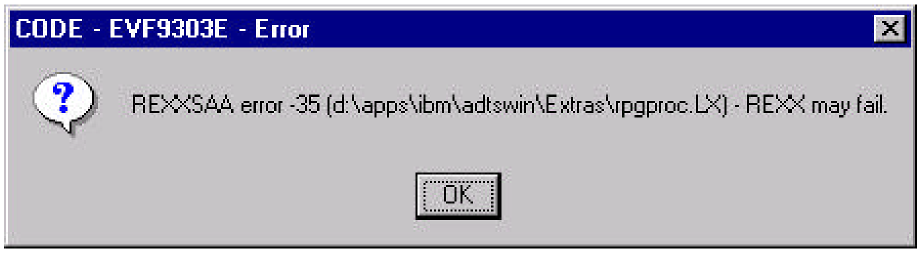
Notice that as soon as you move the cursor away from the first line, the line is highlighted in red indicating that there is a REXX syntax error.


CODE - d:\apps\ibm\adtswin\Extras\rpgprocstep1.lx
File Edit View Actions Options Windows Help Extras
d:\apps\ibm\adtswin\Extras\rpgprocstep1.lx
Row 6 Column 45 Insert

```
00001***************************************************/
00002/* RPGPROC.LX                                      */
00003/*                                                 */
00004/* This macro builds up an RPG procedure           */
00005/* call template                                   */
00006/**************************************************/
```

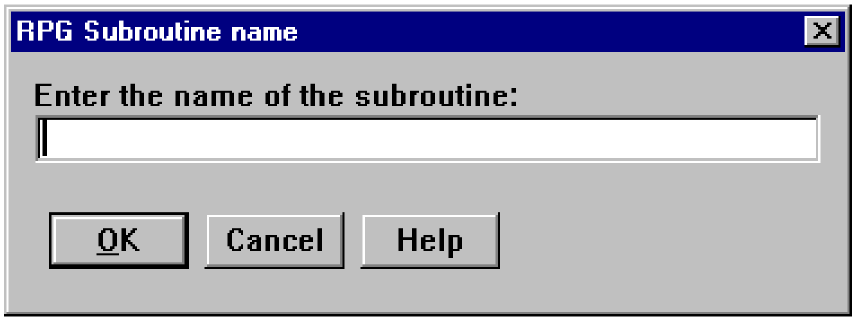
5g. Save the file - this time use the 'Save' icon on the toolbar. It looks like: 💾

Switch to the command line (press the **ESC** key) and type **MACRO RPGPROC**. You will get the following error message that indicates that there is a problem with your REXX program.


CODE - EVF9303E - Error

REXXSAA error -35 (d:\apps\ibm\adtswin\Extras\rpgproc.LX) - REXX may fail.

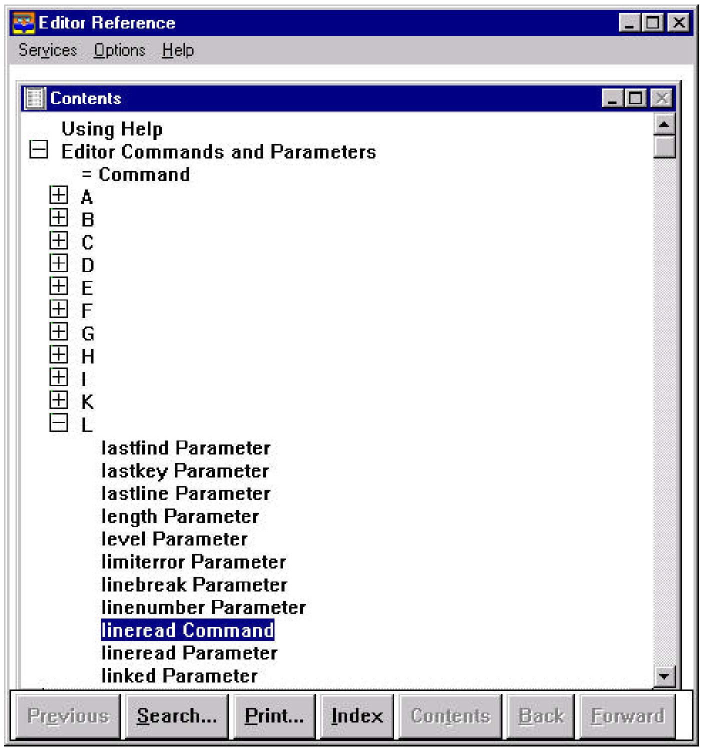
__5h.__ Correct the error by putting a '/' character at the beginning of the first line. Now we will write some REXX code that will show a prompt dialog box that will look like following


RPG Subroutine name

Enter the name of the subroutine:

As a matter of fact, we have already seen similar code in the previous exercise, but at this point it would be very helpful to learn a bit more about the **lineread** editor command.

__5i__. From the '**Help**' editor menu select the '**Editor Reference'** option. The online *Editor Reference* manual comes up. Click on a plus sign next to '**Editor Commands and Parameters'** and then click on the plus sign next to the letter '**L**'. You will see all of the editor commands that start with the letter 'L':



Using Help
Editor Commands and Parameters
= Command
A
B
C
D
E
F
G
H
I
K
L
lastfind Parameter
lastkey Parameter
lastline Parameter
length Parameter
level Parameter
limiterror Parameter
linebreak Parameter
linenumber Parameter
lineread Command
lineread Parameter
linked Parameter

__5j__. The **'lineread Command'** and **'lineread Parameter'** are of immediate interest to us. Double click on each item and carefully read documentation and examples. The following lines of REXX code will setup the dialog box title, a prompt label, and an entry field of length 10:

```
'set lineread.title RPG Subroutine name'
'set lineread.prompt Enter the name of the subroutine: '
'lineread 10 '
```

__5k__. Now that we have understood how to show a dialog box, we still need to figure out how to read the procedure name that user has entered, and which button, either **Ok** or **Cancel)** was pressed. We will not worry about the '**Help**' button for now. You could find out how to do this by reading the Editor Reference for the '**lastline**' and '**lastkey**' commands. Or you could simply use the following two lines:

```
'extract lastline'      /* Read in the text from the entry field */
'extract lastkey'       /* Read in the last key pressed */
```

Once the dialog is dismissed the variable ***lastline*** will contain the procedure name and the variable ***lastkey*** will indicate which button was pressed.
NOTE: The '**Esc**' key corresponds to the '**Cancel**' button press.

__5l__. Some error checking never hurts. Let's make sure that the user actually entered the procedure name and pressed the **Ok** button, otherwise generate an error message.

```
if ((lastline = '') | (lastkey = 'ESC')) then do
  'msg Request cancelled'
  exit
end
```

Notice that we used the *if - then* REXX construct. REXX documentation is available for those who are not very comfortable with the REXX language. From the '**Help**' menu select the '**REXX help'** option. You will find the '**Programming guide'** and '**Reference**' books.

NOTE: We have gathered all the required information from the user, and are ready to create an RPG procedure template. We will use the **insert** editor command and so it is a good idea to read appropriate page of the Editor Reference.

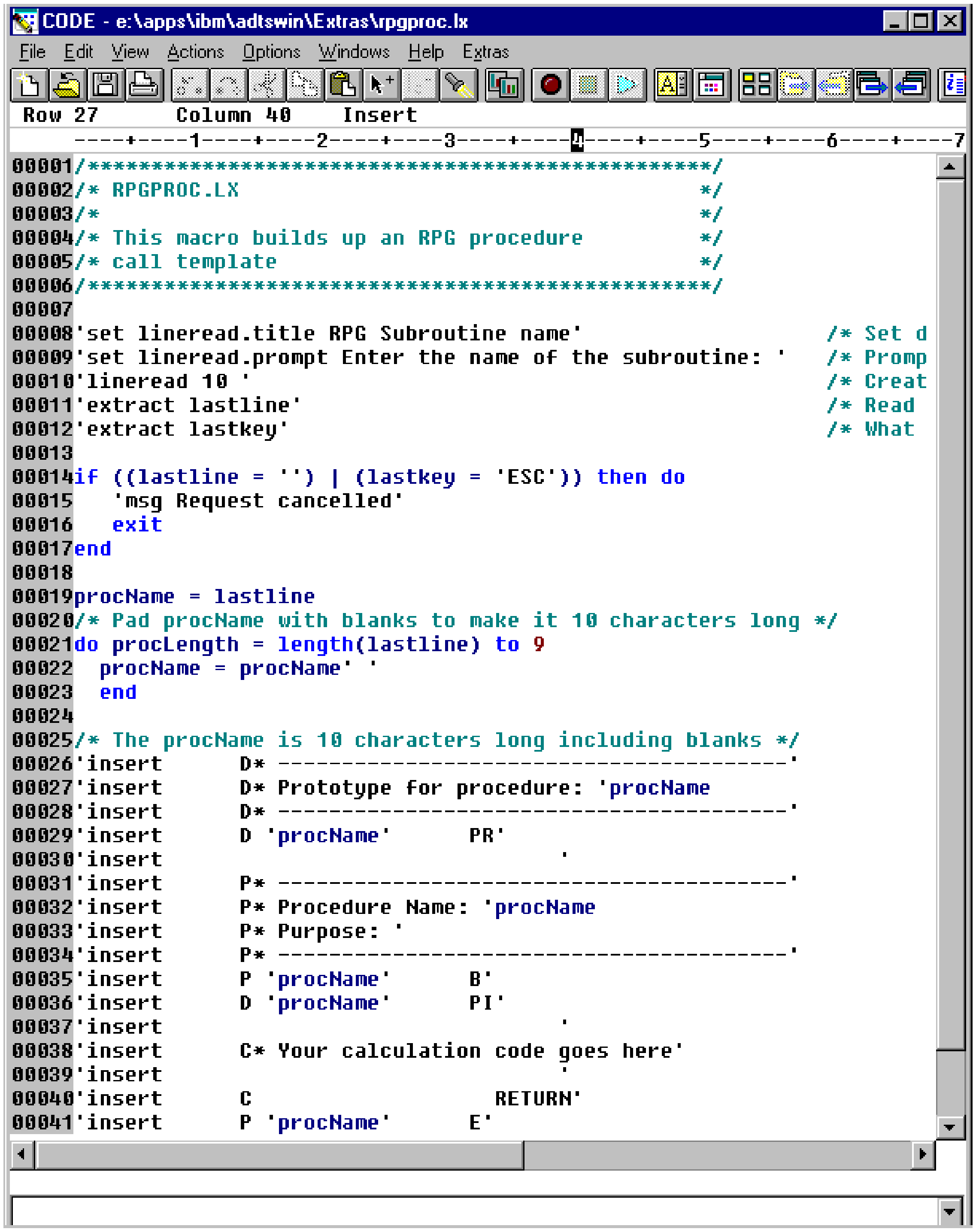
5m. Since RPG is a positional language it is important to make sure that the length of the procedure name variable is no longer than 10 characters. The following code will pad procedure name entered by the user with blanks (up to 10 chars)

```
procName = lastline
/* Pad procName with blanks to make it 10 characters long */
do procLength = length(lastline) to 9
  procName = procName' '
  end
```

5n. Any REXX substitution variables should be placed outside the quotes, while editor commands and strings should be surrounded by single quotes. The final template generation part of the macro will look like:

```
/* The procName is 10 characters long including blanks */
'insert     D* ----------------------------------------'
'insert     D* Prototype for procedure: 'procName
'insert     D* ----------------------------------------'
'insert     D 'procName'      PR'
'insert                        '
'insert     P* ----------------------------------------'
'insert     P* Procedure Name: 'procName
'insert     P* Purpose: '
'insert     P* ----------------------------------------'
'insert     P 'procName'      B'
'insert     D 'procName'      PI'
'insert                        '
'insert     C* Your calculation code goes here'
'insert                        '
'insert     C                  RETURN'
'insert     P 'procName'      E'
```

After putting all the pieces together your code should look like:

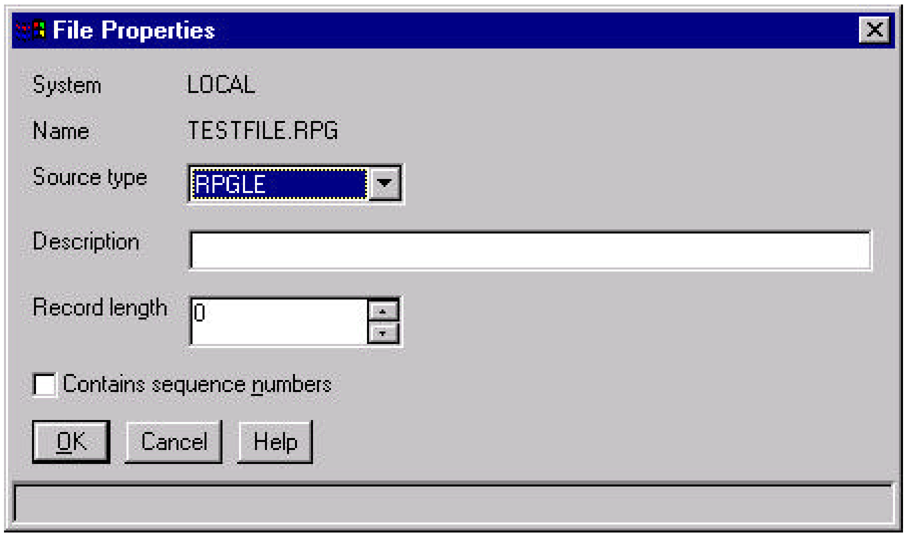

CODE - e:\apps\ibm\adtswin\Extras\rpgproc.lx
File Edit View Actions Options Windows Help Extras
Row 27 Column 40 Insert

```
/*************************************************/
/* RPGPROC.LX                                    */
/*                                               */
/* This macro builds up an RPG procedure         */
/* call template                                 */
/*************************************************/

'set lineread.title RPG Subroutine name'                   /* Set d
'set lineread.prompt Enter the name of the subroutine: '   /* Promp
'lineread 10 '                                             /* Creat
'extract lastline'                                         /* Read
'extract lastkey'                                          /* What

if ((lastline = '') | (lastkey = 'ESC')) then do
   'msg Request cancelled'
   exit
end

procName = lastline
/* Pad procName with blanks to make it 10 characters long */
do procLength = length(lastline) to 9
  procName = procName' '
  end

/* The procName is 10 characters long including blanks */
'insert      D* ----------------------------------------'
'insert      D* Prototype for procedure: 'procName
'insert      D* ----------------------------------------'
'insert      D 'procName'      PR'
'insert                              '
'insert      P* ----------------------------------------'
'insert      P* Procedure Name: 'procName
'insert      P* Purpose: '
'insert      P* ----------------------------------------'
'insert      P 'procName'      B'
'insert      D 'procName'      PI'
'insert                              '
'insert      C* Your calculation code goes here'
'insert                              '
'insert      C                   RETURN'
'insert      P 'procName'      E'
```

Once the file is saved, we are ready to test out the new **RPGPROC** macro!
NOTE: Because executing the macro will actually alter the contents of the current file, it is a good idea to create a brand new local RPG file, say **TESTFILE.RPG** into the editor.
NOTE: If you have not performed **Step 3** of this lab "*Associating name patterns with source types*", please do so now. It is important to make sure that editor views TESTFILE.RPG as an ILE RPG file (the default is OPM RPG)!

5o. On the editor command line type **LX TESTFILE.RPG** and then press **Enter**.
A new file, called TESTFILE.RPG is opened.

5p. To make sure that the CODE editor thinks of it as of an ILE RPG file, bring up the **'File Properties'** dialog from the '**File**' -> '**Properties...**' editor menu.


File Properties

System LOCAL

Name TESTFILE.RPG

Source type RPGLE

Description

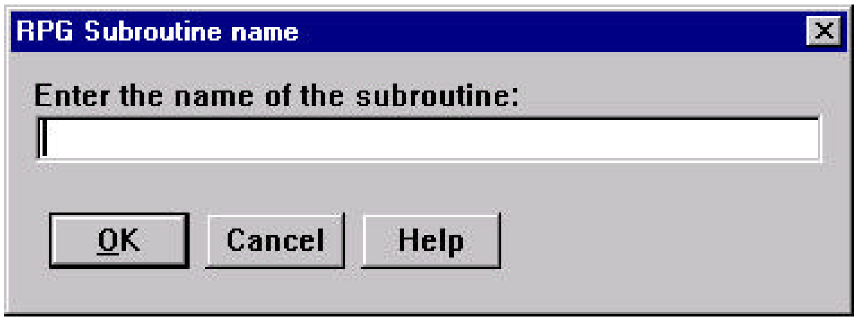
Record length 0

☐ Contains sequence numbers

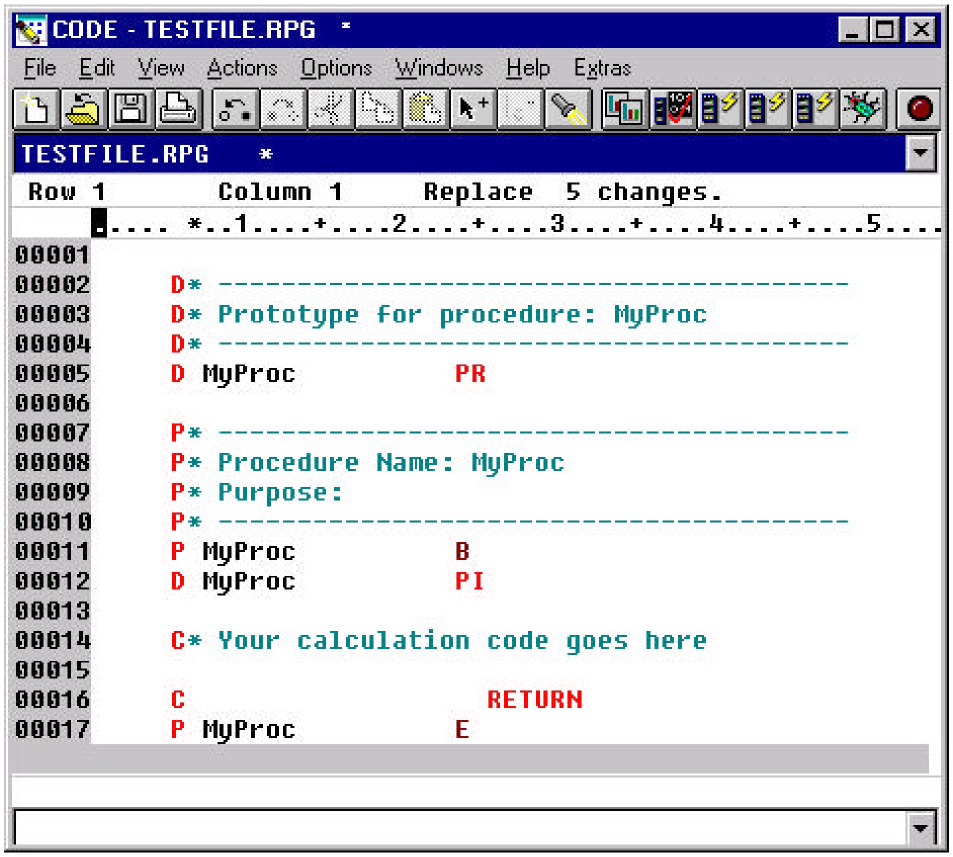
Notice that 'Source type' filed contains RPGLE value. This means that the currently loaded file is an ILE RPG file. If necessary the value could be changed at this point.

5q. Click the 'Cancel' button to dismiss the dialog.

5r. To run the RPGPROC macro, go to the editor command line and type **MACRO RPGPROC** and press the **Enter** button.
The dialog box comes up prompting the user for a procedure name:



5s. Type **MyProc** in the entry field to specify procedure name and then click '**Ok**'. As a result, a procedure template is generated. Notice that the name of the procedure is MyProc. **WOW!**



```
Row 1        Column 1     Replace   5 changes.
     .....  *..1....+....2....+....3....+....4....+....5....

     D* ---------------------------------------------
     D* Prototype for procedure: MyProc
     D* ---------------------------------------------
     D MyProc           PR

     P* ---------------------------------------------
     P* Procedure Name: MyProc
     P* Purpose:
     P* ---------------------------------------------
     P MyProc           B
     D MyProc           PI

     C* Your calculation code goes here

     C                    RETURN
     P MyProc           E
```

Optional exercise - prefilling the procedure name entry field

This exercise is for those who feel fairly comfortable with REXX programming and the editor commands. It's okay to skip this part.

PURPOSE

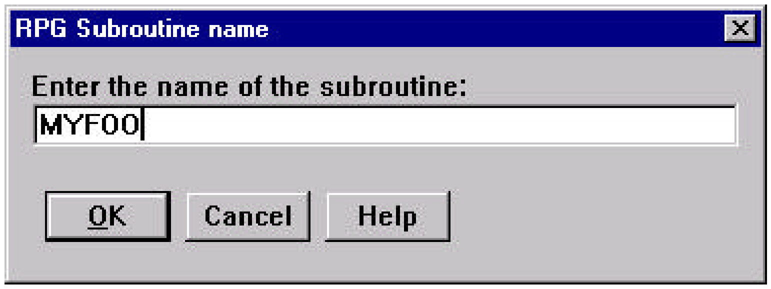
Notice that when the prompt comes up (instruction 5p), the 'Procedure Name' entry field is empty. Sometimes it is useful to prefill it with some default value.

INSTRUCTIONS

Modify your REXX macro so that the 'Procedure Name' entry filed contains value

MYFOO

when the prompt box comes up.


RPG Subroutine name

Enter the name of the subroutine:

MYFOO

HINT

Read 'Editor Reference' book for the **lineread** editor command.

Step 6. Updating editor menu bar

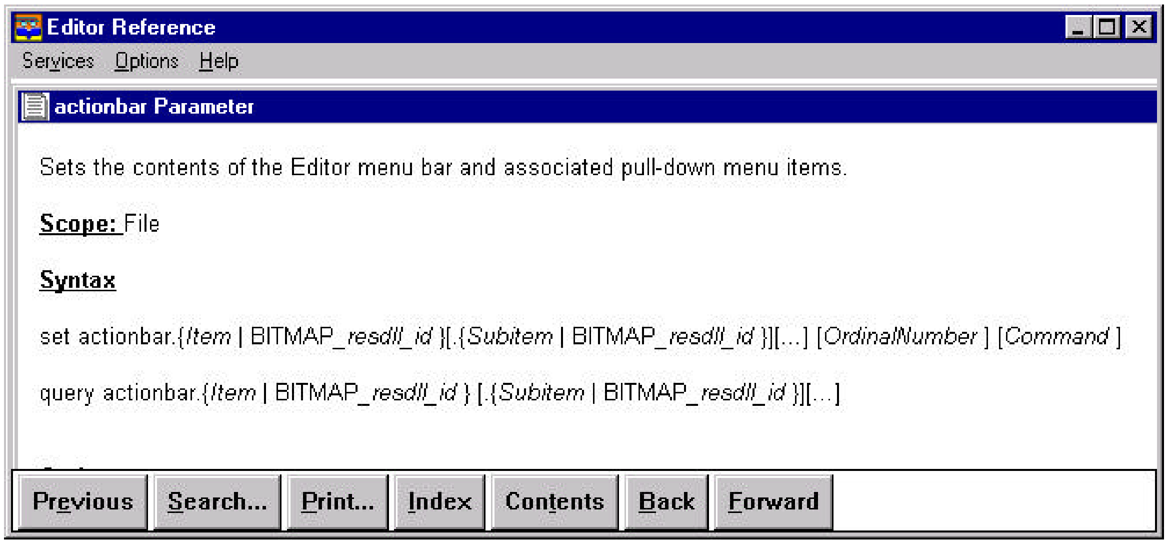
PURPOSE

Once the REXX macro is written you can invoke it from the editor command line. However, for frequently used this may become tedious. In such cases, we can use the editor commands to create new menu items. One of the parameters to the command is the name of your macro. When the menu item is selected, the macro is run.

In this exercise you will create the menu item: '**Extras**' **->** '**COMMON**' **->** '**RPGPROC**'.
You will associate the RPGPROC macro with it and then set the **'Ctrl + Z'** key combination as its shortcut.

INSTRUCTIONS

6a. Use the **ACTIONBAR** editor command to create a new menu item. This is a good time to browse the *'Editor Reference'* book and get familiar with this command.



Sets the contents of the Editor menu bar and associated pull-down menu items.

Scope: File

Syntax

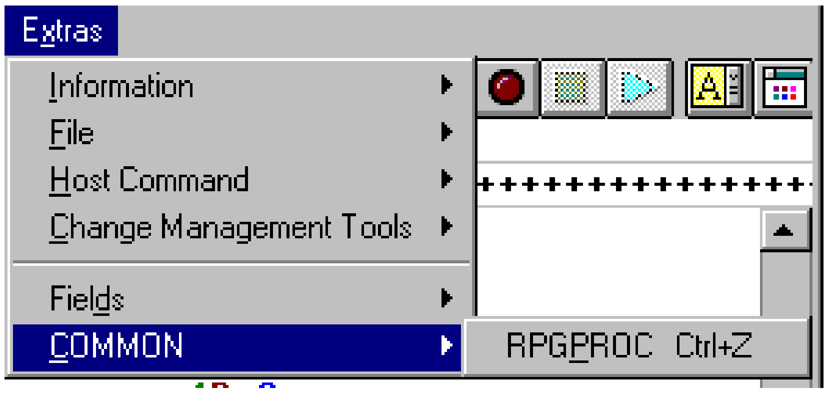
```
set actionbar.{Item | BITMAP_resdll_id }[.{Subitem | BITMAP_resdll_id }][...] [OrdinalNumber ] [Command ]

query actionbar.{Item | BITMAP_resdll_id } [.{Subitem | BITMAP_resdll_id }][...]
```

6b. Switch to the editor command line and type the following command:

SET ACTIONBAR.E~xtras.~COMMON.RPG~PROC\tCtrl+Z MACRO RPGPROC

and press **Enter**.

The resulting menu item will be:



COOL!

NOTE: The '~' character creates a mnemonic for the menu item, while '\t' defines an accelerator key for the menu item. Interestingly enough, **'RPG~PROC'** and **'RPGP~ROC '** are considered to be different menu items.

6c. At this point you can play with the newly created menu item, and the shortcut key. Make sure that they behave the way you expected them to!

Step 7. Updating the editor toolbar and popup menu

PURPOSE

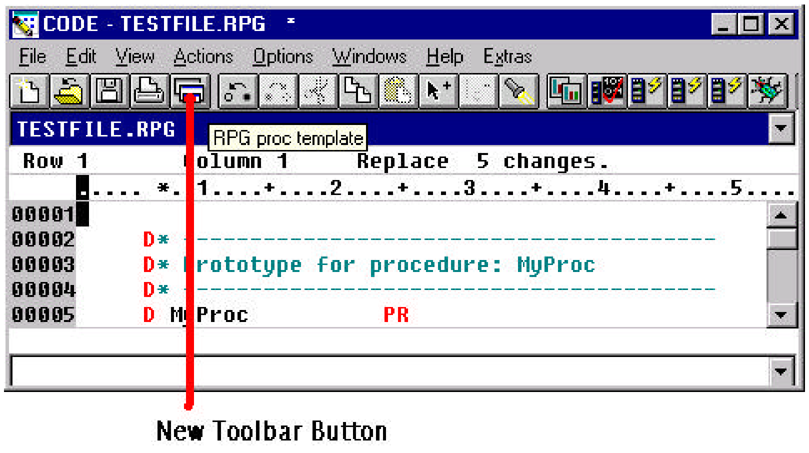
Sometimes programmers like to get fancy and impress their bosses and colleagues. For such occasions, the CODE editor gives you with commands that allow you to update editor's toolbar and popup menu with the items for newly created macros.
In this exercise you will add a new button to the editor's toolbar and a new item to the popup menu. Both of them will again invoke the famous RPGPROC macro.

INSTRUCTIONS

7a. Use the **TOOLBAR** editor command to add a button to the CODE editor toolbar.
Browse the *'Editor Reference'* book to get familiar with this command.

7b. Go to the editor command line and type the following command:
SET TOOLBAR.RPGPROC BITMAP _33 HELP "RPG proc template" 4 MACRO RPGPROC
and then press **Enter**.

The following toolbar item appears in the fifth position from the left:



```
Row 1        Column 1      Replace   5 changes.
      .... *. 1....+....2....+....3....+....4....+....5....
00001
00002      D* -----------------------------------------
00003      D* Prototype for procedure: MyProc
00004      D* -----------------------------------------
00005      D  MyProc           PR
```

Notice that in this example you used the value **_33** for the **BITMAP** option. Bitmaps shipped by CODE/400 are in the range **_1** to **_38** (the underscore character **'_'** is important). Bitmaps can also be loaded from your own *resource DLL*. See the *'Editor Reference'* for more details.

<u>Popup Menu:</u> An example od a popup menu is the menu list that is displayed when the right mouse button is pressed while the mouse pointer is inside the CODE editor. The menu list contains various editing menu items. For example: 'Cut', 'Paste', 'Find selection', etc. This list can be modified by the user. You will do that next.

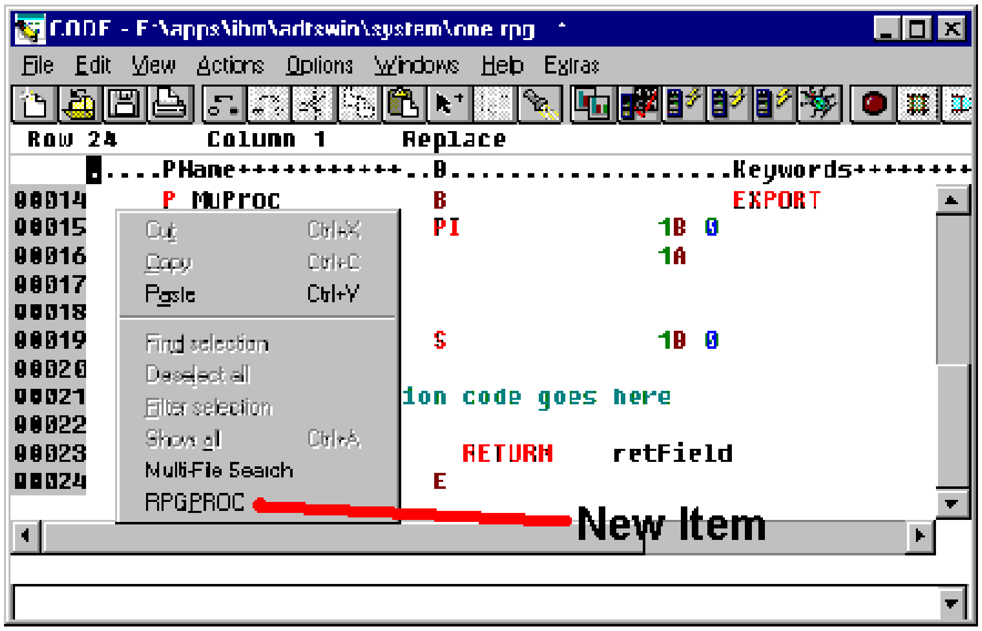
<u>7c</u>. Use the **POPUPMENU** editor command to add items to the CODE editor popup menu. Browse the *'Editor Reference'* book to learn about this command.

<u>**7d.**</u> Go to the editor command line and type the following command:

SET POPUPMENU.RPG~PROC MACRO RPGPROC

and then press **Enter**.

Now when we bring the popup menu the following item will be added:



```
 ....PName++++++++++..B..................Keywords+++++++
00014     P MyProc          B                 EXPORT
00015                       PI               10  0
00016                                        1A
00017
00018
00019                       S                10  0
00020
00021                  ion code goes here
00022
00023                        RETURN    retField
00024                       E
```

<u>**7e**</u> At this point you can play with the newly created toolbar button and popup menu item. Make sure they both behave the way you expected them to! **Cool stuff!**

Step 8. CODESRV - remote execution command

PURPOSE

The CODESRV command is a workstation command that can be used to:

- Get a list of active host CODE servers
- Send commands to the AS/400
- Download and upload source
- Get lists of objects that match a specified filter.

The CODESRV command is just like any other DOS command. You can imbed the command in your files and do all sorts of interesting things.

In order for CODESRV command to become really useful, we must make sure that CODE/400 communication server is started (see **Step 1**).

To see how CODESRV works open up an **MS-DOS Prompt** window and follow the exercises on the next page.

```
C:\>CODESRV_
```

NOTE: In the following exercises when we refer to the library **CODELABxx** you should substitute your workstation number (i.e. 01, 02, 03, ..., etc.) in place of **xx**.

INSTRUCTIONS

8a. To see the list of active CODE/400 servers type:

CODESRV SERVER

at MS-DOS prompt and then press the **Enter** key. Your list will probably only have **OS400** in it.

8b. To print the MSTDSP source member using SEU, at MS-DOS prompt type:

CODESRV EXEC OS400 STRSEU OPTION(6) SRCFILE(CODELABxx/QDDSSRC) SRCMBR(MSTDSP)

8c. To list all the source members in CODELABxx/QDDSSRC type:

CODESRV LIST OS400 "CODELABxx/QDDSSRC(*)"

The result should be:

EMPMST MSTDSP PRJMST REFMST RSNMST End of file or list.

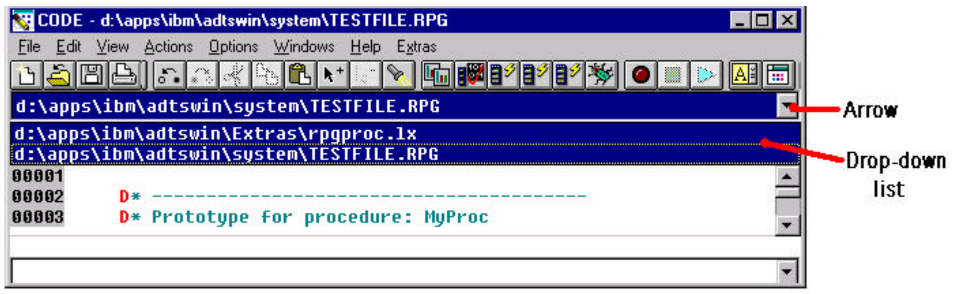
8d. Type **CODESRV ?** to get to the help for the command. If you are really ambitious use the **CODESRV GET OS400...** and **CODESRV PUT OS400...** to download and upload members from the AS/400. Notice from the help that you can also use the CODESRV command to shut down all servers (you can have up to ten connections at a time) or a specific connection to a server.

NOTE: You can also invoke CODE/400 tools from the AS/400. The simplest way is to create a user-defined option in PDM. For example, to invoke the CODE Editor on a source member you would use the following syntax:

CALL QCODE/EVFCFDBK PARM('37' 'Y' 'OS400' '<LOCAL> CODEEDIT "<server>lib/file(member)"')

Switching between files:

Multiple files can be loaded into the CODE/400 editor simultaneously. In order to switch from one file to another there is a drop-down list which is located directly under the toolbox. Once you click on the down arrow on the right, the entire list shows up and you can select the file.


CODE - d:\apps\ibm\adtswin\system\TESTFILE.RPG
File Edit View Actions Options Windows Help Extras
Arrow
Drop-down list

```
d:\apps\ibm\adtswin\system\TESTFILE.RPG
d:\apps\ibm\adtswin\Extras\rpgproc.lx
d:\apps\ibm\adtswin\system\TESTFILE.RPG
00001
00002      D* -----------------------------------------
00003      D* Prototype for procedure: MyProc
```

More Importantly:
The CODESRV command can be used in your macros to execute remote commands! Let's take a closer look at a macro called SEUPRINT which uses the CODESRV command in order to print a current member being edited on the host.

<u>8e</u>. From the editor command line run the LX SEUPRINT.LX command.
The file SEUPRINT.LX is loaded into the editor:

```
/* SEUPRINT - macro to print the current member being edited on the host. It uses the*/
/*                     SEU print option.                                              */

/* Blank out the message line */
'msg' '   '

/* Get full name of file being edited */
'extract name'

/* Get the name of the server, file and member */
parse var name '<' server '>' fn '(' mn ')'

/* Drop /ADM from server name if it exists */
parse var server host '/' junk

/* Issue error if this is a LOCAL file... */
if host = 'LOCAL' then do
  'msg Host Print is not valid for local files.'
  'ALARM'
  exit
end

/* Prompt user to save source, then print it on host... */
'SAVEALL PROMPT START CODESRV EXEC 'host' STRSEU SRCFILE('fn')
SRCMBR('mn') OPTION(6) (LOG'

'msg Member printed using STRSEU. See Command Shell for status.'
```

Notice that the CODESRV command has been used to submit the SEU print option (OPTION (6) in this case) to the AS/400 host.

Step 9. CODE/400 editor profiles

PURPOSE

The menu items, toolbar buttons, and shortcuts that you created in previous exercises will only work for the current edit session. If you open a different file or start a new edit session the menu items will not exist and the shortcuts will do nothing. To make these changes to the editor more permanent you can use '*profiles*'. A profile is nothing more than a text file containing editor commands. Some of the profiles supplied with the editor provide specific editing features and run automatically at specific times.

Profile	When does it run?	Can I change it ?
PROFINIT.LXU	When the editor starts.	Yes
PROFSYS.LXU	Just before each file is loaded.	Yes
xxx.LXL; xxx = cbl, rpgle400, etc.	After PROFSYS.LXU, but before a file of type xxx is loaded.	No
xxx.LXU	After xxx.LXL but before the file is loaded.	Yes. Add your own xxx specific commands here.
PROFILE.LX	The last profile run before each file is loaded.	Yes
xxx.LXS	Whenever a file of type xxx is saved.	Yes

We will take a closer look at the RPGLE400.LXL profile, and will create an RPGLE400.LXU profile, adding all of our menu and toolbar button creation commands to it.

INSTRUCTIONS

9a. From the editor command line execute the **LX RPGLE400.LXL** command to load a file **RPGLE400.LXL** into the editor.

9b. Look through the file. It contains various editor commands that run once the ILE RPG file gets loaded into the editor. Let us take a closer look at some of them:

```
/* initial fonts settings */
'SET FONT.A BLACK/WHITE          "Page"'
'SET FONT.B GREY/WHITE           "Line"'
'SET FONT.C BRIGHT RED/WHITE  "Spec"’
                         ........................................
                         ........................................
```

Setup initial fonts for various language constructs...

```
'SET FULLPARSE SUBMIT READ STOP "Parsing file" ILEPAR ALL'
'SET PARSER ILEPAR'
```

Parse the file using ILEPAR parser type...

```
'SET ACTIONBAR.LP_VIEW.S~how. 2 ;'
'SET HELP. 16054'
'SET ACTIONBAR.LP_VIEW.S~how.~Control ;INCLUDE CONTROL;SET E
```

Create some menu items...

```
'SET ACTIONPREFIX.F ;SET PREFIXENTRY;ILEPAR Q'
'SET ACTIONPREFIX.F? ;SET PREFIXENTRY;ILEPAR O'
'SET ACTIONPREFIX.P ;SET PREFIXENTRY;ILEPAR PROMPT'
                              .....................................
                              .....................................
```

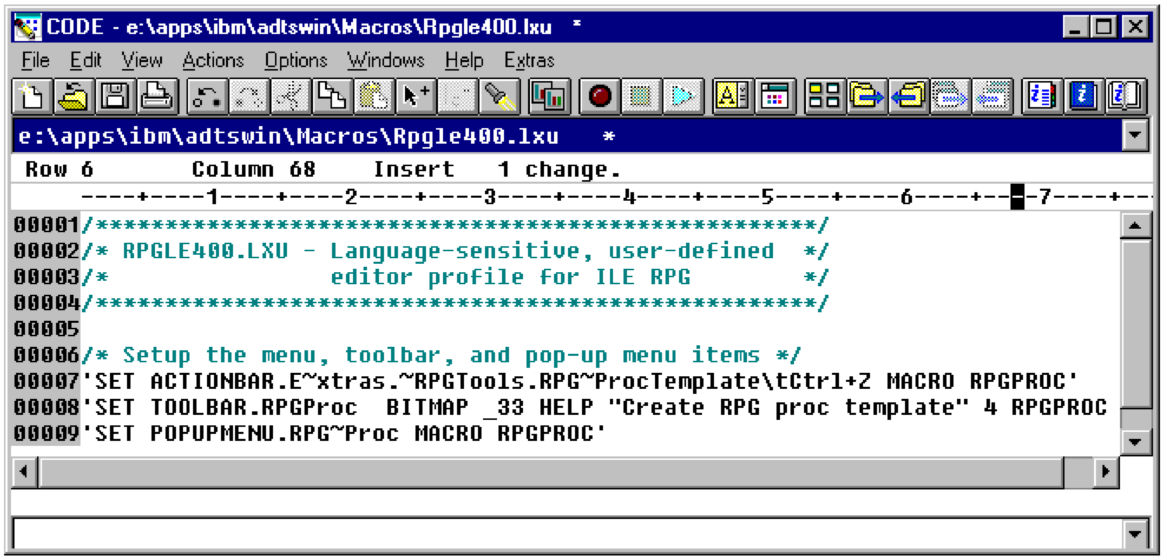
Create ILE RPG specific prefix area commands.

9c. At this point we will be creating an RPGLE400.LXU profile. It runs after RPGLE400.LXL, but before an ILE RPG file is loaded. We will use this profile to add the menu options and toolbar buttons associated with the RPGPROC macro whenever an ILE RPG file is loaded! On the editor command line type:

LX RPGLE400.LXU

and then press the **Enter** key.

9d. Add the following familiar lines to the file and save it.


CODE - e:\apps\ibm\adtswin\Macros\Rpgle400.lxu *
File Edit View Actions Options Windows Help Extras
e:\apps\ibm\adtswin\Macros\Rpgle400.lxu *
Row 6 Column 68 Insert 1 change.

```
/*****************************************************/
/* RPGLE400.LXU - Language-sensitive, user-defined  */
/*                editor profile for ILE RPG         */
/*****************************************************/

/* Setup the menu, toolbar, and pop-up menu items */
'SET ACTIONBAR.E~xtras.~RPGTools.RPG~ProcTemplate\tCtrl+Z MACRO RPGPROC'
'SET TOOLBAR.RPGProc  BITMAP _33 HELP "Create RPG proc template" 4 RPGPROC
'SET POPUPMENU.RPG~Proc MACRO RPGPROC'
```

9e. Close all of the editor windows using the '**File**' -> '**Exit**' menu option.

9f. Bring up an **MS-DOS Prompt** window and run the following command:

CODEEDIT COMMON.RPG

which brings up an editor and loads the COMMON.RPG file.

The menu items, popup menu item and toolbar button associated with the RPGPROC macro are available now. The RPGLE400.LXU profile that you just created ran just before we loaded the ILE RPG file!

NOTE: It is not a good idea to make changes to the **xxx.LXL** files because they get replaced once the workstation is updated to the new release of CODE/400. On the other hand, **xxx.LXU** files are left untouched and hence your changes 'survive' the CODE/400 update!

9g. Close CODE editor.

This sections of the lab is complete!

The Lab - Section 2: Lpexlets

Section Introduction

In this section we will learn how to program the CODE/400 editor using the Java language. Java is an object oriented programming language that is, compared to other OO languages like C++, relatively “easy to digest”. Over the course of the past few years a large number of Java - related terms have emerged:

- Java Beans
- Cookies
- Applets
- Servlets

So, not to fall far behind, CODE/400 added its own Java - related term: **Lpexlets.**
They are extensions to the CODE editor written in Java that allow a much richer set of GUI components than REXX macros. In this section we will write a very simple Lpexlet that provides the GUI interface for the RPGPROC macro. The Lpexlet will only take care of gathering the information from the user and will then call a REXX macro to generate an RPG procedure template. (The REXX part has already been implemented in the previous section).
To run your Lpexlet, on the editor command line type: **RUNJAVA Lpexlet_Class_Name.**

As a CODE/400 user, Java applies to you in the following ways:

- Today, as a language that helps you customize CODE/400 editor via Lpexlets.
- Today, as a programming language for your client user interfaces.
- Tomorrow (V4R2 and beyond), as a programming language on the AS/400 itself.

Java Applets

Java can be used to write applets, which are small programs that can only run inside web browsers such as **Netscape Navigator** or **Microsoft Internet Explorer**. These are mini-programs, but they have full user interface capabilities. They run right inside the browser. Java is traditionally an interpreted language, like **Visual Basic** and **Smalltalk**, and the web browsers today all include a Java interpreter engine.

Java applets can be used inside a traditional **HTML** (*HyperText Markup Language*) web page to add logic, graphics or user interaction. They can even be used to access data from a host, such as **DB2/400**.

The key things to remember about applets are:

- They only run inside a browser. They have no “main window” of their own, but rather use the real estate of the web browser.
- They physically live on the same server as the web page itself. The web browser, upon encountering an HTML “**APPLET**” tag inside the HTML source for a web page will return to the server to retrieve the applet (as pointed to by the **APPLET** tag), and download it into memory where it will be run.
- They are not permitted to access the local client’s hard drive or run programs on the local client. They are also not allowed to communicate back to any host server except the one they came from (the restrictions can be waived with “signed” applets that are run by consenting users).

Java applets can target AS/400 data and programs. This can be done using built-in Java communications support for TCP/IP sockets programming, or it can be done using the **AS/400 Toolbox for Java** set of classes written by IBM Rochester. This Java code offers a significantly easier means to access AS/400 services than raw communications coding.

Java Applications

While the early excitement around Java was due to its unique ability to program web pages with live code, this is not Java’s only role. It is also a full fledged application programming language, and can be used effectively to write full applications, which are invoked from the command line as with traditional language applications.

Using Java to write applications offers all the functionality and portability benefits of Java applets, but:

- Removes the security “sandbox” restrictions that applets have.
- Does not offer, yet, the exceptional benefit of being loaded on demand that applets enjoy. This means distribution and maintenance are bigger considerations, for *client* Java applications.

NOTE: the **AS/400 Toolbox for Java** code can be used for Java applications or applets;
the **AS/400 Toolbox for Java** classes are shipped with CODE/400.

To run a Java application on a particular operating system, you must have a Java Virtual Machine (JVM - interpreter) on that operating system. All current operating systems have now, or will soon have, a JVM built into them.
The Java Development Kit (JDK) is required to develop Lpexlets. The JDK or Java Runtime Environment (JRE) is required to run them. Both are available from JavaSoft's web site *www.javasoft.com*.

Code/400 - Advanced topics: Hands On Lab

You will:

- Create an **RPGProc** Java class that extends the **LpexCommand** class - a must for every Lpexlet.
- Create another new class called **RPGProcFrame**, that extends **JFrame** which is a Java-supplied class for putting up a dialog and implements a Java-supplied interface for handling GUI events.
- Compile Java classes using the CODE/400 Java class generation mechanism.
- Write an RPGPROCJAVA macro that reads in data provided by the Lpexlet and generates an RPG procedure template.
- Run your Lpexlet from the CODE/400 editor and see the results.
- Play with the ‘RPG Procedure’ SmartGuide.

This lab is not intended to teach you how to program in Java, however, we will give you pointers about relevant language constructs along the way. So, if you see **Java Reference** and **END Java Reference** tags, that is where you find Java language bits.

Ready? Let us continue our journey of CODE/400 Lpexlets...

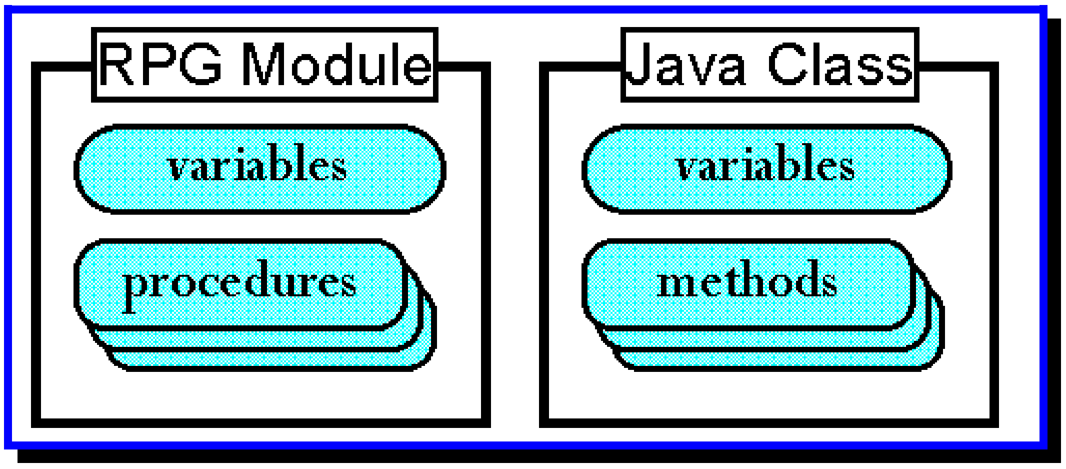
Step 1. Creating an RPGProc Lpexlet Class.

Java Reference:

- **Comments** in Java come in two forms:
  - **Multiple line**: These start with "`/*`" and continue until an ending "`*/`" pair is found.
  - **Single line**: To put a comment on a line or end of a line, start it with `//`
- **Classes**. These, like AS/400 ILE RPG modules, allow you to divide your source code into functions (methods in Java, procedures and subroutines in RPG) and variables those functions need. These are typically self-contained groupings. Classes contain multiple fields (variables) and methods.
- **Methods**. These, like AS/400 ILE RPG procedures and subroutines, contain all the actual code your program or application will use. Unlike RPG, in Java executable code can only exist in methods. And methods can only exist inside classes.

What is a **class**? It is a key construct in Java: *all* code and *all* variables exist *only* inside classes. In fact, code must exist inside methods which must exist inside classes.

Java classes are similar to ILE RPG IV **modules**! Modules contain variables and RPG procedures and subroutines. Java classes contain variables and methods. ***Methods*** are like RPG ***procedures***



A class in Java typically looks like this:

```
public class MyClass
{
  // variables
  // methods
}
```

NOTE: the keyword **class**, and the braces delimiting the beginning and end of the class. In this example, "`MyClass`" is the user-supplied name of the class. The Java keyword **public** indicates this class is accessible by everyone. This is an optional keyword - without it only other classes in *this* package have access to this class.

- **Inheritance**. One of the main features of every Object Oriented language is the ability to easily extend already existing code. In Java, this feature is implemented by the means of *Inheritance*. You can write a class (call it *BaseClass*) that provides some basic services. (By **services** I mean Java methods or ILE RPG procedures/subroutines). If a new class that you are implementing (call it *SophisticatedClass*) needs to provide the same basic services, and perhaps even more, *SophisticatedClass* can **inherit** all basic services from the *BaseClass*, and only implement new functionality.
  In Java we use the **extends** keyword to indicate the inheritance. Here is a typical example:

```
public class SophisticatedClass extends BaseClass
{
   // variables
   // methods
}  // end SophisticatedClass
```

- **Polymorphism** is another cornerstone concept of Object Oriented languages. When your *SophisticatedClass* inherits from the *BaseClass* there maybe some methods implemented by the *BaseClass* whose behavior you would like to alter. You can **override** a method. If your BaseClass provides a method *MyMethod( ),* your *SophisticatedClass* can also implement *MyMethod( )* which behaves differently then the inherited one. At run time Java decides which method to use appropriately. This feature of Java language is called **polymorphism**.

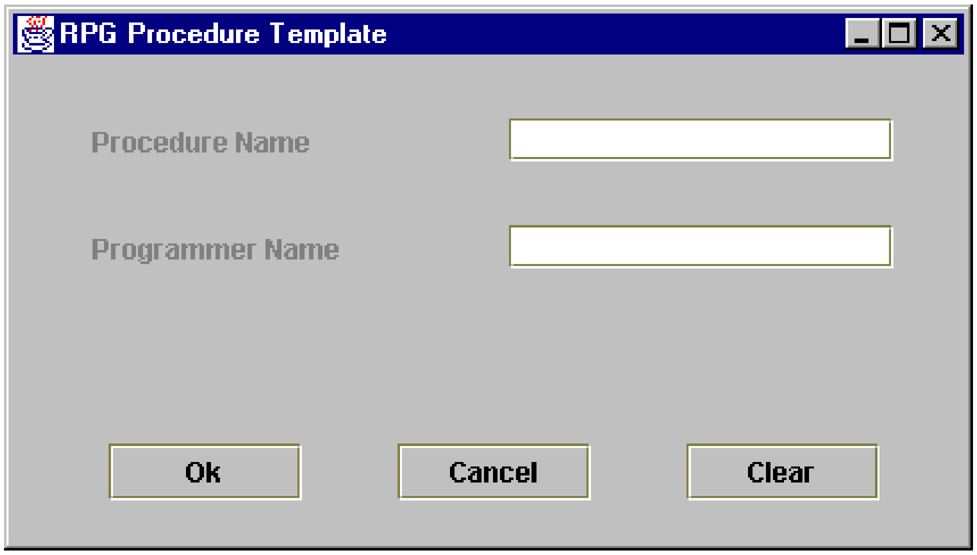
END Java Reference

PURPOSE

CODE/400 ships a set of Java classes. Information is available from the **'Help' ->** '**Java help' ->** '**Lpex Java readme'** menu option. Note that you have to open a Java file for **'Java help'** option to be available. One of the classes that CODE/400 ships is called *LpexCommand* class. This class is your interface to writing Lpexlets. In this section we will implement an **RPGProc** class that will inherit from the *LpexCommand* class, as must every Lpexlet. In addition, every Lpexlet must override the method *lpexEntry( )* - a main entry point into the Lpexlet. This method gets called by the CODE editor when **'RUNJAVA Lpexlet_Class_Name'** command is run.

In our case Lpexlet_Class_Name will be RPGProc and hence the command becomes **'RUNJAVA RPGProc'**. Remember the class name is case sensitive!

The RPGProc Lpexlet will put up a nice dialog prompting the user for the Procedure Name and the Programmer Name.



Once all information is entered the Lpexlet will call a REXX macro to generate the procedure template. The reason is very simple - we already have code that does this job. So we will reuse a part of RPGPROC macro.

INSTRUCTIONS

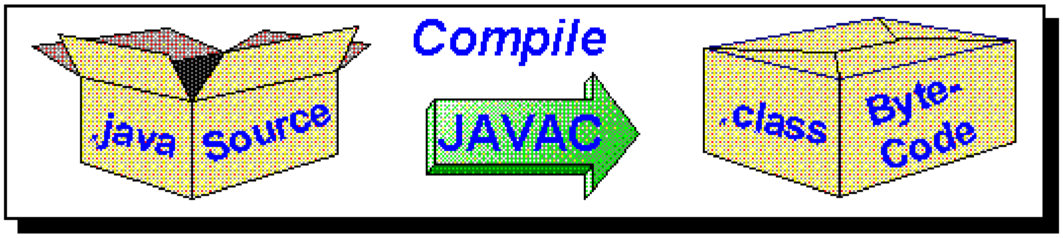
1a. Start up the CODE editor and create a new file named **RPGProc.java**

1b. Below is the code for the RPGProc class.

```
import RPGProcFrame;

public class RPGProc extends LpexCommand
{
    static RPGProcFrame rpgProcFrame = null;

    /*  lpexEntry() - main entry point from LPEX.  Overrides LpexCommand's.  */
   public static int lpexEntry (String arg)
   {
     if( rpgProcFrame == null )
       rpgProcFrame = new RPGProcFrame();

     rpgProcFrame.setVisible(true);
     return 0;
   } // end lpexEntry()


   // Once the Ok button is pressed, need to set DOCVARs
   public static int setDocVars(String procName, String pgmrName)
   {
      lpexCommand("SET DOCVAR.PROCNAME " + procName);
      lpexCommand("SET DOCVAR.PGMRNAME " + pgmrName);

      lpexCommand("MACRO RPGPROCJAVA");
      return 0;
   } // end setDocVars()

   /*  lpexNotify() - tell LPEX to notify us on exit.
   public static int lpexNotify()
   {
      return LPEX_NOTIFY_EXIT;
   }  // end of lpexNotify()

   /*  lpexExit() - we're being terminated, dispose of the toolbar  */
   public static int lpexExit (String arg)
   {
      rpgProcFrame.dispose();          // get rid of the dialog
      return 0;
   }  // end of lpexExit

} // end class RPGProc
```

Java Reference:

Typically you have only one class per source file (**.java**). And the name of the class coincides with the name of the source file (not counting the **.java** extension). Then the source file will be compiled into one *ByteCode* (**.class**) file with the same name as the class. The compiler is called **JAVAC** and it converts source into easily interpreted *ByteCode*.



CODE/400 automates this compilation step, just like for any other supported language. We will see this feature later in this lab.

- **Objects**. These are "*instances*" of classes, and are necessary to use classes that contain non-static methods or variables. They are created by defining a variable, specifying the class as the type, and equating the variable to an *instance* or *allocation* of the class using the **new** operator in Java.

- **Instance variables**. These are non-static variables declared at the class level and available to all methods in the class. Each instance (object) of the class gets its own copy of these variables. Compare to global variables in RPG.

- **Local variables**. These are variables declared inside a method and are local to that method. They are only "alive" as long as the method is running.

- **Constructors**. These are special methods that each class can optionally have that are called by Java when the class is first "*instantiated*" (an instance is allocated). They are used to initialize variables and state, similar to RPG's ***INZSR** subroutine. They are identified by their name - it is the same as the class

END Java Reference

NOTE: The **import** statement in Java is like **COPY** in RPG. Hence **import RPGProcFrame** means that the file RPGProcFrame.java (which probably defines an RPGProcFrame class) is included into our *RPGProc.java* file. As a matter of fact, the **RPGProcFrame** class defines the user interface part of this Lpexlet. We will develop this class in Step 2 of this section.

NOTE: In our implementation of the lpexEntry( ) function (remember that every Lpexlet has to override this function!) we create a new RPGProcFrame object and then make it visible using *setVisible( )* method.

NOTE: We will create a *setDocVars( )* method which will be called by the RPGProcFrame class. We will then use the *lpexCommand( )* method of the LpexCommand class in order to execute the CODE editor commands. In order to pass the values of the procedure and programmer name to the REXX macro we need to save these values in the editor variables. They will be retrieved later by the REXX macro:

```
lpexCommand("SET DOCVAR.PROCNAME " + procName);
lpexCommand("SET DOCVAR.PGMRNAME " + pgmrName);
```

Last but not least we will use the lpexCommand( ) function to call the REXX macro **RPGPROCJAVA**. This macro - a shortened version of RPGPROC - will be implemented later in this lab.

Help for the LpexCommand class is available from **'Help' ->** '**Java help' ->** '**LpexCommand help '** menu option.

1c. Enter all Java source into **RPGProc.java** file.

NOTES ABOUT TYPING:

- ***Case is important***. Java names are case sensitive. "`MyVar`" does ***not*** equal "`myvar`".
- ***White space is not important.*** Leave/insert as many blanks as you like.
- ***Watch for the semi-colons*** (;) at the end of executable lines of code! They are important.

1d. Save your file in the C:\ADTSWIN\JAVA directory by going to the editor command line and typing:

```
SAVE C:\ADTSWIN\JAVA\RPGProc.java
```

and then pressing the **Enter** key. Remember that the file name is case sensitive!

Step 2. Creating the "RPG Procedure Template" dialog box - RPGProcFame class.

PURPOSE

In the lpexEntry( ) method of the RPGProc class we create an rpgProcFrame object of type RPGProcFrame that is responsible for putting up the dialog box. Now is the time to implement the RPGProcFrame class.

Java Reference.

Some Java-supplied classes

RPGProcFrame class will inherit from the class **JFrame**. JFrame is a Java-supplied class. It is responsible for putting up the dialog window and window's border. Other Java-supplied classes that are used by the RPGProcFrame class are:

- **JPane**. Object of this class fills in the space provided by the JFrame. It also looks after the placement of all other user interface components.
- **JButtion**. Objects of this class are pushbuttons. (**Ok**, **Cancel**, and **Clear** in our case).
- **JLabel**. Objects of this class are text labels.
- **JTextField**. Objects of this class are entry fields where the user types in the input.

Interfaces

Many Object Oriented languages provide the ability to inherit services from multiple classes. This feature is called **multiple inheritance**. Due to some efficiency and complexity considerations, Java does not directly support multiple inheritance. However, every once in a while, a need for such construct arises. To overcome this difficulty, Java supports a concept similar to a class, called an **interface**. An interface does not provide services, it only defines them. A class can **implement** an interface. Implementing an interface, means implementing all services/methods that a particular interface defines. A class can extend another class and implement interfaces at the same time. Here is a typical example:

```
public class SophisticatedClass extends BaseClass
                               implements BaseInterface
{
   // variables
   // methods
}  // end SophisticatedClass
```

Event Driven Programming in GUI Systems

In RPG you display a screen by writing to one or more record formats, and retrieve data entered by the user by reading a record format. Reading a display file will return data in the fields and indicators (which indicate which key was pressed). This is *Screen-driven* programming. Your program writes and reads screens of information.

In GUI environments, it is different. Your program gets "*notified*" of every single user action - pressing a key, pushing a button, moving the mouse, etc.. These actions are called ***events***. Your program can choose to process individual events or let the system do its default action for them (usually nothing). This is called *event-driven* programming

Event Driven Programming in Java

In Java, "events" are Java objects (instances of Java classes) that are sent to your own class ***if you tell Java to***!

How do I tell Java to send events to my class?

You have to do **three** things (don't do these yet, just read):

1. Indicate that your class is capable of responding to these events by including the code "**implements xxxListener**" on the class definition, where xxx indicates the events you want to be informed of. For example, "**implements ActionListener**" will cause the system to inform you of ***action*** events (versus say, ***typing*** events or ***mouse move*** events).

2. Supply a method in your class that will be called for specific events. These methods have to use the *exact* names and parameter types that Java defines for each event. For example, for action events it requires the method **"public void actionPerformed(ActionEvent event)"**.

3. For each GUI component, such as a push button, after creating it you must "register" that it is to send its events to your class. Do this using the **"addActionListener**(*instance-of-your-class*)**"** method that all input-capable Java components support.

END Java Reference.

INSTRUCTIONS

2a. In the CODE/400 editor open a new file **RPGProcFrame.java**

2b. The next few pages contain source code for the RPGProcFrame class.

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;

/* RPGProcFrame.java This class creates and handles the UI for the RPGProc Lpexlet */
public class RPGProcFrame extends JFrame implements ActionListener
{
    private JPanel contentPane = null;
    private JButton cancelButton = null;
    private JButton clearButton  = null;
    private JButton okButton     = null;
    private JLabel pgmrNameLabel = null;
    private JLabel procNameLabel = null;
    private JTextField pgmrNameTextField = null;
    private JTextField procNameTextField = null;

    /* RPGProcFrame class constructor */
    public RPGProcFrame()
    {
      super();
      setSize(426, 240);
      setTitle("RPG Procedure Template");

     // Create Ok button object
      okButton = new JButton("Ok");
      okButton.setBounds(42, 170, 85, 25);
      okButton.addActionListener(this);

      // Create cancel button object
      cancelButton = new JButton("Cancel");
      cancelButton.setBounds(169, 170, 85, 25);
      cancelButton.addActionListener(this);

      // Create clear button object
      clearButton = new JButton("Clear");
      clearButton.setBounds(296, 170, 85, 25);
      clearButton.addActionListener(this);
      // ------------------------------------------------------------------
```

```
  // Create text label for procedure nanme
  procNameLabel = new JLabel("Procedure Name");
  procNameLabel.setBounds(35, 27, 146, 20);

  // Create text label for programmer name
  pgmrNameLabel = new JLabel("Programmer Name");
  pgmrNameLabel.setBounds(35, 74, 147, 20);
  // ----------------------------------------------------------------

  // Creating an entry field for procedure name
  procNameTextField = new JTextField();
  procNameTextField.setBounds(218, 27, 169, 19);

  // Creating an entry field for programmer name
  pgmrNameTextField = new JTextField();
  pgmrNameTextField.setBounds(218, 74, 169, 19);
  // ----------------------------------------------------------------

  // Construct the JPanel object - client canvas and add all controls
  contentPane = new JPanel();
  contentPane.setLayout(null);
  // ----------------------------------------------------------------

  // Add all entry controls and corresponding Labels to the client pane
  contentPane.add(procNameLabel, procNameLabel.getName());
  contentPane.add(pgmrNameLabel, pgmrNameLabel.getName());
  contentPane.add(procNameTextField, procNameTextField.getName());
  contentPane.add(pgmrNameTextField, pgmrNameTextField.getName());

  // Add all button controls to the client pane
  contentPane.add(okButton, okButton.getName());
  contentPane.add(cancelButton, cancelButton.getName());
  contentPane.add(clearButton, clearButton.getName());
  // ----------------------------------------------------------------

  // ----------------------------------------------------------------
  // Now that everything is constructed, set the client pane to contentPane
  // ----------------------------------------------------------------
  setContentPane(contentPane);
  // ----------------------------------------------------------------

} // end constructor()
```

```
   /**
    * Override actionPerformed( ) method of the ActionListener interface
    * If any registered button is pressed, this method gets invoked
    */
    public void actionPerformed(ActionEvent evt)
    {
      // First of all figure which button was just pressed
      String arg = evt.getActionCommand();

      if( arg.equals("Ok") )              // OK button is pressed
      {
        // Update DOCVARs to be used by the REXX macro
        RPGProc.setDocVars(procNameTextField.getText(),
                           pgmrNameTextField.getText());
        dispose();                        // close the dialog
      } // end if(Ok button is pressed)
      else if( arg.equals("Cancel") ) // Cancel button is pressed
      {
        dispose();                        // close the dialog
      } // end if(Cancel button is pressed)
      else if( arg.equals("Clear") )   // Clear button is pressed
      {
        procNameTextField.setText("");   // Clear the procNameTextField
        pgmrNameTextField.setText("");  // Clear the prmrNameTextField
      } // end if(Clear button is pressed)
    } // end actionPerformed()

} // end class RPGProcFrame
// -----------------------------------------------------------------
```

NOTE: As we pointed out before, this lab is not intended to teach you the Java language. But we still would like to highlight a few key points.

- The RPGProcFrame class inherits from the Java-supplied **JFrame** class and implements the Java-supplied **ActionListener** interface.
- The RPGProcFrame class implements only two methods: a class constructor *RPGProcFrame( )* and *actionPerformed( ).*

REMEMBER: A CONSTRUCTOR IS A METHOD THAT HAS THE SAME NAME AS THE CLASS, AND HAS NO RETURN TYPE.

In the class constructor we create the dialog window, all dialog controls, and place these controls inside the dialog window. We also “register” all buttons with our RPGProcFrame class. Whenever a button is pressed, an event is sent to the RPGProcFrame class.

```
// Make sure client is listening to the button press events
okButton.addActionListener(this);
cancelButton.addActionListener(this);
clearButton.addActionListener(this);
```

Note: “**this**” is a special Java built-in keyword that represents the current instance of the current class. So, for example, a reference to an instance variable, as in **`x=10`** is equivalent to **`this.x=10`**

The **actionPerformed( )** method is defined by the **ActionListener** interface. Since the RPGProcFrame class **implements** the ActionListener interface, it must provide an implementation of this method. Whenever a button is pressed, an event is sent to the RPGProcFrame class and an *actionPerformed( )* method gets called. We figure out which button: ‘Ok’, ‘Cancel’, or ‘Clear’ caused the event to be generated, and act accordingly...

2c. Enter all the Java source into **RPGProcFrame.java** file.

2d. Save your file in the C:\ADTSWIN\JAVA directory by going to the editor command line and typing:

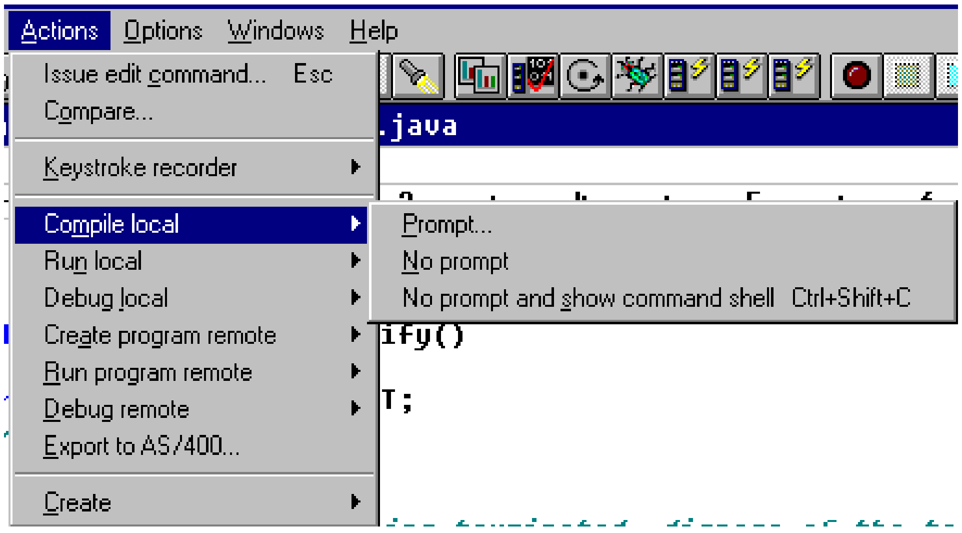
SAVE C:\ADTSWIN\JAVA\RPGProcFrame.java

and then pressing the **Enter** key. Again, the file name is case sensitive!

Step 3. Using CODE/400 to compile your Java classes.

PURPOSE

The CODE editor provides a set of Verify/Compile/Debug actions for any supported AS/400 language including Java. However, Java classes can run on your PC and on your AS/400. CODE/400 targets both: one for Lpexlet development and the other for AS/400 Java development. We therefore provide two sets of Compile/Run/Debug actions: local and remote.

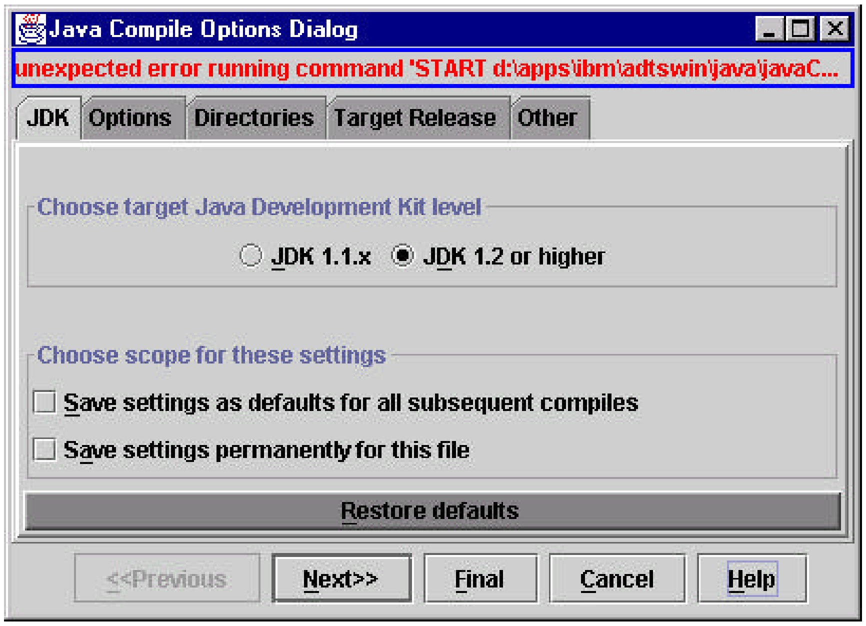


In this exercise we are developing Lpexlets and will therefore concentrate on local actions.

INSTRUCTIONS

3a. Make sure your current file is **RPGProcFrame.java**.

3b. From the '**Actions**' editor menu select the **'Compile local' -> 'Prompt...'** option.
After a few seconds (be patient - this is Java) the following dialog comes up.


Java Compile Options Dialog

unexpected error running command 'START d:\apps\ibm\adtswin\java\javaC...

Choose target Java Development Kit level

( ) JDK 1.1.x [✓ (●) JDK 1.2 or higher]

Choose scope for these settings

☐ Save settings as defaults for all subsequent compiles

☐ Save settings permanently for this file

This dialog has several pages of Java compiler settings. You can use the ‘Next>>’ and ‘Previous>>’ buttons to navigate between pages. Get familiar with the dialog. You will need to use it quite a bit once you get into serious Lpexlet development!

<u>3c</u>. The defaults are just fine for now. Press the ‘**Final**’ button and watch how RPGProcFrame class gets compiled. You will notice a ‘Compling...’ message in the editor message area (just above the editor command line).

NOTE: Once the compile is completed, and if no errors are detected, you will get a ‘Compiled clean’ message in the editor message area. If your Java class contains errors, an ‘Error list’ window comes up indicating all of the compile errors. Double clicking on an error message takes you to the line that causes the problem.

<u>3d.</u> In the CODE/400 editor switch to the **RPGProc.java** file.

<u>3e.</u> This time we will use a no prompt compile option. From the ‘Actions’ menu select **‘Compile local’ -> ‘No prompt’** option and watch the RPGProc class compiling.

Now all of your Java classes are compiled and **.class** files are generated. Wasn’t that easy?

WOW!

Step 4. Creating the RPGPROCJAVA macro and running the Lpexlet!

PURPOSE

We are almost ready to test out our first Lpexlet but there is still one piece of the puzzle that is still missing. Remember, we need to call the **RPGPROCJAVA** macro to generate the procedure template. As a matter of fact, we can reuse most of the REXX code from the RPGPROC macro. After that, the testing stage begins!

INSTRUCTIONS

4a. Open a new file RPGPROCJAVA.LX by typing: **LX RPGPROCJAVA.LX** on the editor command line and then pressing the **Enter** key.

4b. The REXX code on the next page should look very familiar. The only trick is the use of two DOCVARs:

```
/* Read in the DOCVARs that are set by the Lpexlet */
'EXTRACT DOCVAR.PROCNAME INTO 'procName
'EXTRACT DOCVAR.PGMRNAME INTO 'pgmrName
```

Remember, we did a ‘SET DOCVAR’ in the RPGProc class? The ‘EXTRACT DOCVAR‘ is how we retrieved values stored in the DOCVARs. This is the data exchange mechanism between Lpexlets and REXX macros.

4c. Type in the following REXX code and save the file:

```
/**************************************************/
/* RPGPROCJAVA.LX                                  */
/*                                                 */
/* This macro builds up an RPG procedure call      */
/* template.                                       */
/* It uses RPGProc Lpexlet for prompting...        */
/**************************************************/

/* Read in the DOCVARs that are set by the Lpexlet */
'EXTRACT DOCVAR.PROCNAME INTO 'procName
'EXTRACT DOCVAR.PGMRNAME INTO 'pgmrName

/* Pad procName with blanks to make it 10 characters long */
do procLength = length(procName) to 9
  procName = procName' '
  end

/* The procName is 10 characters long including blanks */
'insert     D* ---------------------------------------'
'insert     D* Prototype for procedure: 'procName
'insert     D* ---------------------------------------'
'insert     D 'procName'      PR'
'insert                       '
'insert     P* ---------------------------------------'
'insert     P* Procedure Name: 'procName
'insert     P* Purpose:        '
'insert     P* Written by:     'pgmrName
'insert     P* ---------------------------------------'
'insert     P 'procName'      B'
'insert     D 'procName'      PI'
'insert                       '
'insert     C* Your calculation code goes here'
'insert                       '
'insert     C                 RETURN'
'insert     P 'procName'      E'

'TRIGGER FULLPARSE'
```

All the pieces are now ready and we can now test the Lpexlet.

4d. Open a new ILE RPG file COMMON.RPG by typing: **LX COMMON.RPG** on the editor command line and then press the **Enter** key.

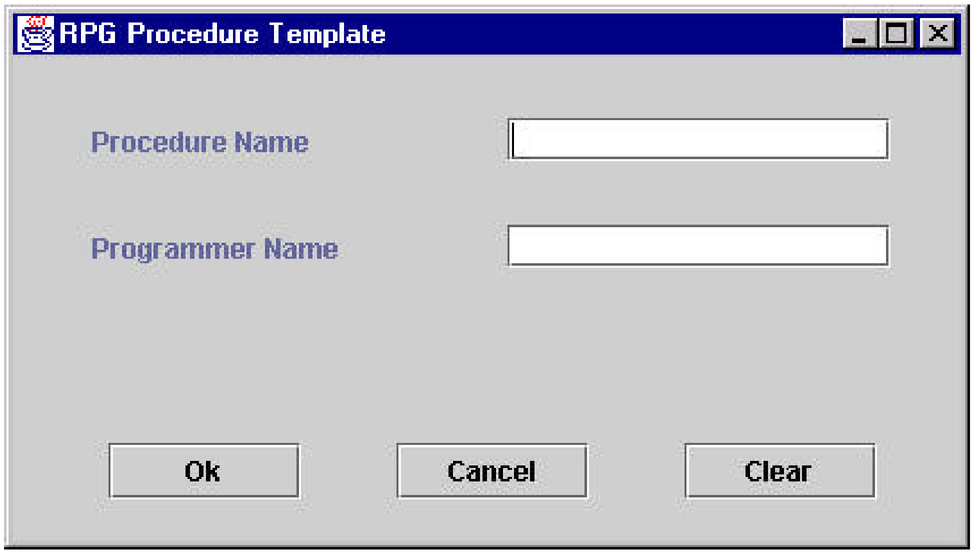
4e. Go to the editor command line and type:

RUNJAVA RPGProc

and then press the **Enter** key.

Note the **case is important** when you call Java class.

The following Java dialog comes up prompting the user for the procedure name and the programmer name:



W O W!!!

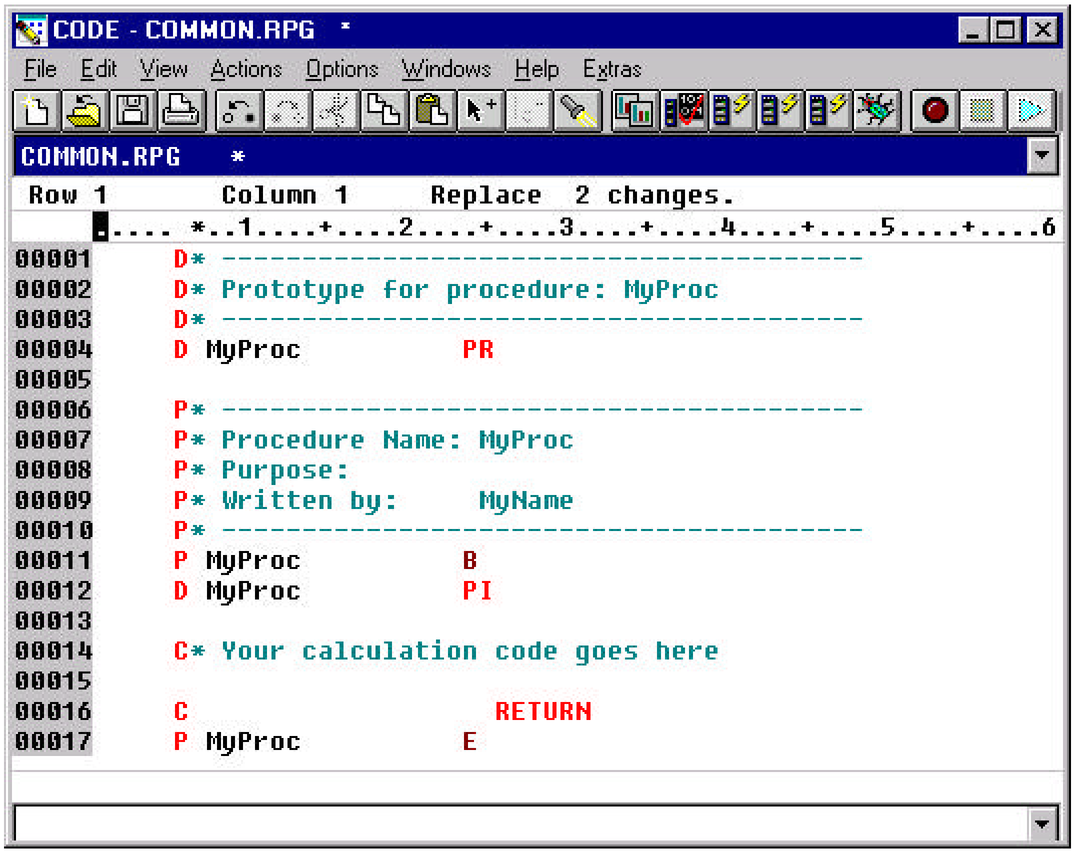
4f. Enter the following values in the entry fields:

In Procedure Name field enter: **MyProc**

In Programmer Name field enter: **MyName**

and press the **Ok** button.

The resulting procedure template is shown on the next page:


CODE - COMMON.RPG *
File Edit View Actions Options Windows Help Extras
COMMON.RPG *
Row 1 Column 1 Replace 2 changes.

```
     *...1....+....2....+....3....+....4....+....5....+....6
     D* ----------------------------------------
     D* Prototype for procedure: MyProc
     D* ----------------------------------------
     D MyProc          PR

     P* ----------------------------------------
     P* Procedure Name: MyProc
     P* Purpose:
     P* Written by:     MyName
     P* ----------------------------------------
     P MyProc          B
     D MyProc          PI

     C* Your calculation code goes here

     C                   RETURN
     P MyProc          E
```

Notice that generated template is very similar to the one created by the RPGPROC macro. This time, however, the template also contains the programmer's name. It would be fairly easy to add other entry fields to the existing dialog to prompt user for other important pieces of information.

<u>**4g**</u>. Close the CODE editor. From the **'File'** menu select '**Exit**'.

<u>*** Done the Lab! ****</u>

For a two hour lab, you have done very well! This CODE editor programming tour may have left you bewildered, but you got this far - congratulations. Soon enough you will impress your boss and colleagues with some cool extensions to the CODE editor!

See you around the water cooler!

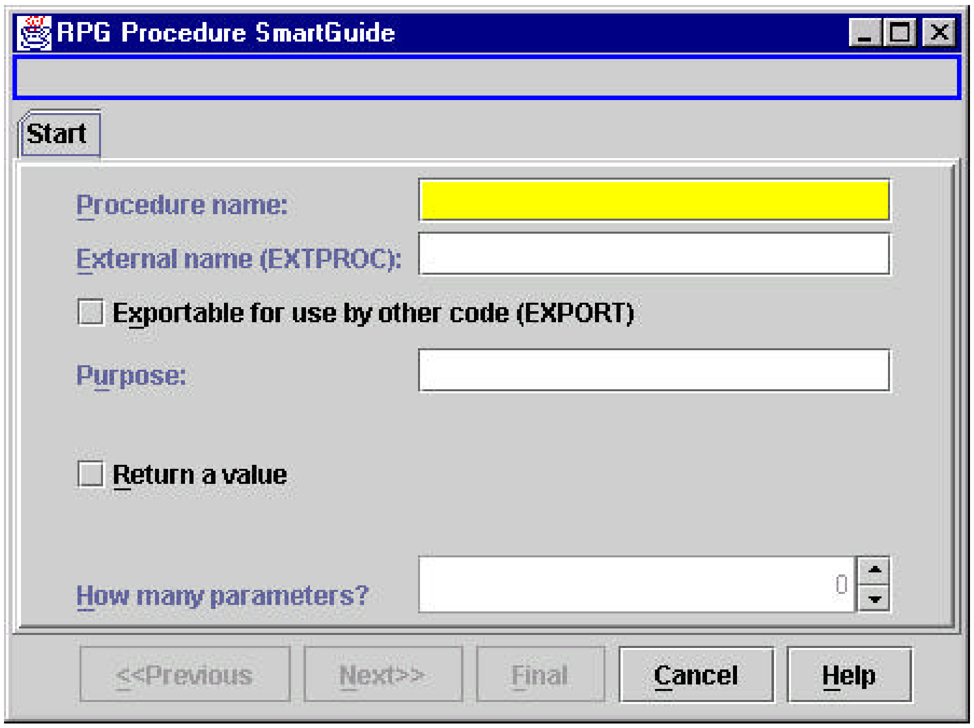
Appendix - The RPG Procedure SmartGuide

This section is not part of the two hour core lab. We just want to show you how fancy you can get with Lpexlets. CODE/400 ships a Java- based SmartGuide framework The documentation is available from the editor '**Help**' menu: '**Java help' -> 'SmartGuide framework'**.

One of the samples that comes with CODE/400 is a SmartGuide to generate an RPG procedure template.

a. Open an ILE RPG file (you can even use COMMON.RPG).

b. From the '**Actions**' menu select **'SmartGuides' -> 'Create Procedure...'.** The following dialog comes up:



Procedure name:
External name (EXTPROC):
☐ Exportable for use by other code (EXPORT)
Purpose:
☐ Return a value
How many parameters? 0

Notice how additional pages appear if you increase the number of parameters or indicate that the procedure has a return value. Entry fields colored in yellow must be filled in, the others are optional.

Play with the SmartGuide, have fun, and good luck with CODE/400!