IBM WebSphere Development Studio client for iSeries

CODE - Selected Advanced Topics

Session Id 404595
Agenda Key 36LF

Inge Weiss and the iSeries Team

iweiss@ca.ibm.com

IBM Toronto Laboratory

Version 4 SP 2

homepage: http://www.ibm.com/software/ad/iseries
newsgroup: news://news.software.ibm.com/ibm.software.code400

Eighth Edition (September, 2002)

The information contained in this document has not been submitted to any formal IBM test and is distributed on an "as is" basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Comments concerning this notebook and its usefulness for its intended purpose are welcome. You may send written comments to:

IBM Canada Ltd.

8200 Warden Avenue, Markham, Ontario, L6G 1C7

Attention: Inge Weiss, CODE Advanced Topics

or e-mail to**: iweiss@ca.ibm.com**

Technical Information

For more technical information on CODE/or WebSphere Development Studio client for iSeries contact either
Dave Slater at slater@ca.ibm.com
Claus Weiss at weiss@ca.ibm.com

Education

CODE courses:

S6186   CODE/400 for iSeries - Basic (2 days)
S6205   CODE/400 for iSeries - Advanced (1 day)

S6286   iSeries Application Development using WDSc for iSeries - Basic (2 days)

Trademarks

IBM is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation.

iSeries
DB2/400
ILE
Integrated Language Environment
IBM
OS/400
RPG/400
Visual Age
WebSphere

Trademarks of other companies as shown

'Intel'   'Intel Corporation'

'Microsoft'   'Microsoft Corporation'

'Windows'   'Microsoft Corporation'

Overall Lab Guide

The objective of the lab CODE - Selected Advanced Topics is to explore some of the customization possibilities that are available in the CODE Editor. At the end of the lab, the student should know how to create REXX macros, menu items, toolbar buttons, and popup menus. The Lab also shows how to make these changes persistent by adding them to the appropriate editor profile. Part two of this lab shows how to create an Lpexlet, an extension to the editor written in Java.

The lab also gives the student an opportunity to work with the Remote Systems Explorer, learn how to invoke the CODE tools from there and, as an optional exercise, learn how to create filters and user actions.

The exercises should be completed in sequence.

Note: *The pictures in these labs show similar tasks. Some of the names and directories may be different from the environment you are working with.*

Prerequisites

The participants should be familiar with CODE. They should be able to use the CODE Editor. Also, it is helpful if the student is familiar with basic MS Windows operations such as working with the desktop and basic mouse operations such as opening folders and performing drag-and-drop operations.

Table of Contents

Table of Contents

<u>Introduction</u>

The **IBM WebSphere Development Studio for iSeries** product is a suite of e-business enabling technologies including host and workstation components:

Host Components
* ILE RPG
* ILE COBOL
* ILE C
* ILE C++
* Application Development ToolSet (ADTS) includes PDM, SEU, SDA, RLU

Workstation Components (WebSphere Development Studio client, WDSc)
* WebSphere Studio Site Developer Advanced

    + iSeries Plugins
    * Remote Systems Explorer
    * WebFacing
* Classic Tools
    * CODE
    * VisualAge RPG

The **CoOperative Development Environment**, better known as **CODE,** 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 iSeries; and completely organize your programming projects.

The CODE 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 iSeries. 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 iSeries 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, printer file reports and physical files easy and fun.

• **IBM Distributed Debugger**

A source-level debugger that allows you to debug an application running on a host iSeries 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 together and allows you to quickly access all the power of CODE and to effectively manage and organize your development projects.

The functionality of CPO is being replaced by the WebSphere Studio workbench. In this lab you will have a chance to use the Explorer perspective, instead of CPO to reflect this fact.

The Goal

In this session, you will learn some nontrivial features and functionality of the CODE tools. You 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 you can be with CODE even when there is no connection to the iSeries host. We are confident that CODE 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 iSeries. Now let's spend a couple of hours playing and see if you agree.

The Tool

Installing CODE

The CODE tool of the WebSphere Development Studio client for iSeries (WDSc) product consists of two parts:

1. The 'back-end' which resides on the iSeries.
   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 WDS product.
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 iSeries (assuming that the workstation files have been copied into the iSeries ifs).
   The workstation install uses the Windows Installer.

The minimum hardware requirements for CODE are an Intel® Pentium II processor or faster with 128MB of memory, a SVGA 800 x 600 monitor, CD-ROM drive, and a mouse or pointing device. The recommended workstation hardware when using the WebSphere Studio workbench with the CODE tools is a workstation with 256MB of memory and a SVGA 1024 x 768 monitor. An install of **WDSc** uses about 1.4GB 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 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 (RPG and DDS).
- For column-sensitive languages there is the CODE FIELDS ON command that simplifies text insertions and deletions.
- On-line language reference help.

CODE Editor Programming (ultimate customization)

Despite its rich functionality, the CODE editor may still lack features that suit the needs of a particular iSeries 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 toolbar.
- 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.

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

Now let's begin our journey into the wonderful world of CODE...

Step 0. Connecting to the iSeries

PURPOSE

Communication between the iSeries and your workstation can be configured for:
- •TCP/IP 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 Express; Personal Communications; or RUMBA to handle the communications.

For this lab session, you will use TCP/IP communications. On the workstation, the CODE daemon needs to be running in order to allow TCP/IP communication with the iSeries. When your PC is restarted after the CODE installation, the CODE daemon is started for you. If you closed the daemon or want to start it manually, you can do so from Start → Programs → IBM Websphere Development Tools for iSeries → Communications → Communication Daemon.

INSTRUCTIONS

1a. Ensure that the Communication daemon is running.
This program waits and listens for an iSeries to contact
it on a specific TCP/IP port and then makes a connection.
You should see an icon in your system tray (bottom right of your screen).
You can interact with CODE communications by using the pop-up menu of this icon.
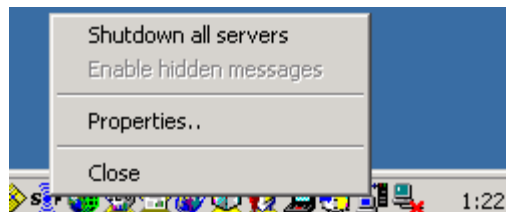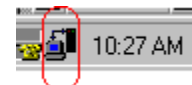
1b. Start a 5250-emulation session.

1c. Sign on to the iSeries. Your userid and password should both be **WDSCLABxx** where **xx** is your workstation number (01, 02, etc.).

> **Note:**
> Instead of the **Enter** key you may have to use the **Ctrl** key in your 5250-emulation session.

1d. At the iSeries command line type:

STRCODE RMTLOCNAME(*RESOLVE) CMNTYPE(*TCPIP)

The ***RESOLVE** keyword will get the IP address of your work station and with this information the **CODE server** will communicate with the **CODE daemon** that runs on your workstation.

You should see a screen that has **EVFCLOGO** in the upper left-hand corner. The CODE server is now active and waiting for commands from the workstation.
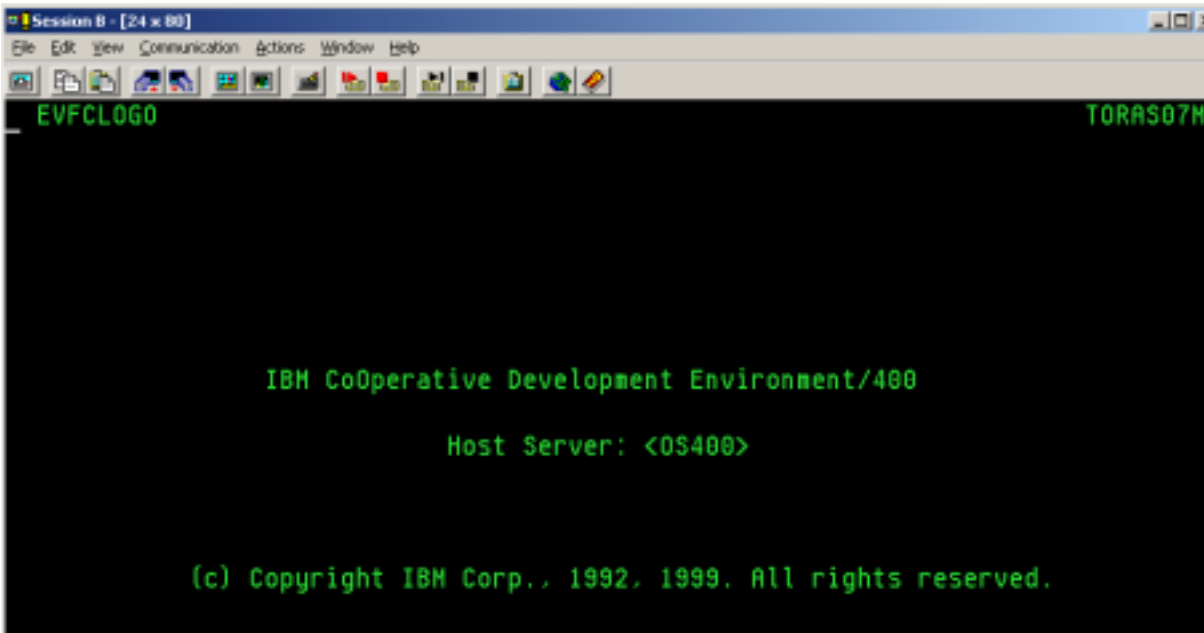
There is a different way to start the CODE server; just for your information we describe it here as well.

If you know your workstations IP hostname, you could specify it directly in the STRCODE command:

STRCODE RMTLOCNAME(PC_hostname) CMNTYPE(*TCPIP)

```
                        Start CODE (STRCODE)

Type choices, press Enter.

Host server name . . . . . . . .   OS400        Character value
Remote location name . . . . .   PC hostname
Communications type  . . . . .   *TCPIP         *PRV, *APPC, *TCPIP
```
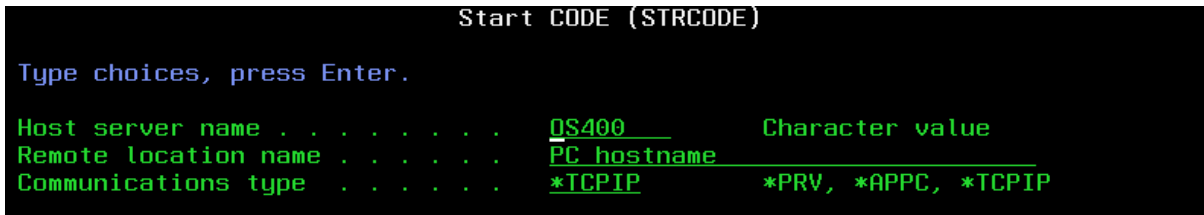
The PC_hostname would be either the TCP/IP host name of your PC or its dotted IP address in quotes. For example '9.21.99.99'. You can determine your PC host name, by typing :
   hostname
 at a DOS command line.

> ## Note:
> If you are still on an OS/400 version lower than V5, the ***Resolve** keyword is not supported, you could instead use the **STRCODETCP** command.
>
> This will call a CL program which automatically figures out which IP address your emulator is using and invokes the STRCODE command that is shipped with the product. This CL program can be found in the QCLSRC source file in the CODELABxx library.

Step 1. The Remote Systems Explorer (RSE)

CODE invocation choices

The new WDSc product with its workbench gives the programmer some more choices how to invoke the editor. You have the choice to either work with the CODE tools in a stand alone environment, or you could use the new WebSphere Studio workbench capabilities of accessing iSeries objects directly and then invoke the CODE tools from the workbench.

Using the WebSphere Studio workbench to invoke the CODE Editor

In this Lab we will use the WebSphere Studio workbench to access iSeries objects, and then start the CODE tools from the workbench.

Before you start using WebSphere Development Studio Client, you will need to reset it. This will delete any changes to the environment due to previous use of the tool on the PC you are working on.

The instructor will tell you how to reset the WebSphere Development Studio Client environment by removing the **WORKSPACE** folder from the **WDSC** folder.

> **Warning:**
> If you work on your own system, be careful as this will remove all project information stored in the workbench. You will loose your work if you reset the workbench!

After you have reset the workbench go ahead and start WDSc

Invoking WebSphere Development Studio client (WDSc)

> To start Development Studio client,
>
> - Press the **Start** button on the task bar of your desktop
> - Choose *Start→Programs→IBM WebSphere Development Studio Client for iSeries -> IBM WebSphere Studio Site Developer Advanced*

- After a few moments of loading, the workbench appears
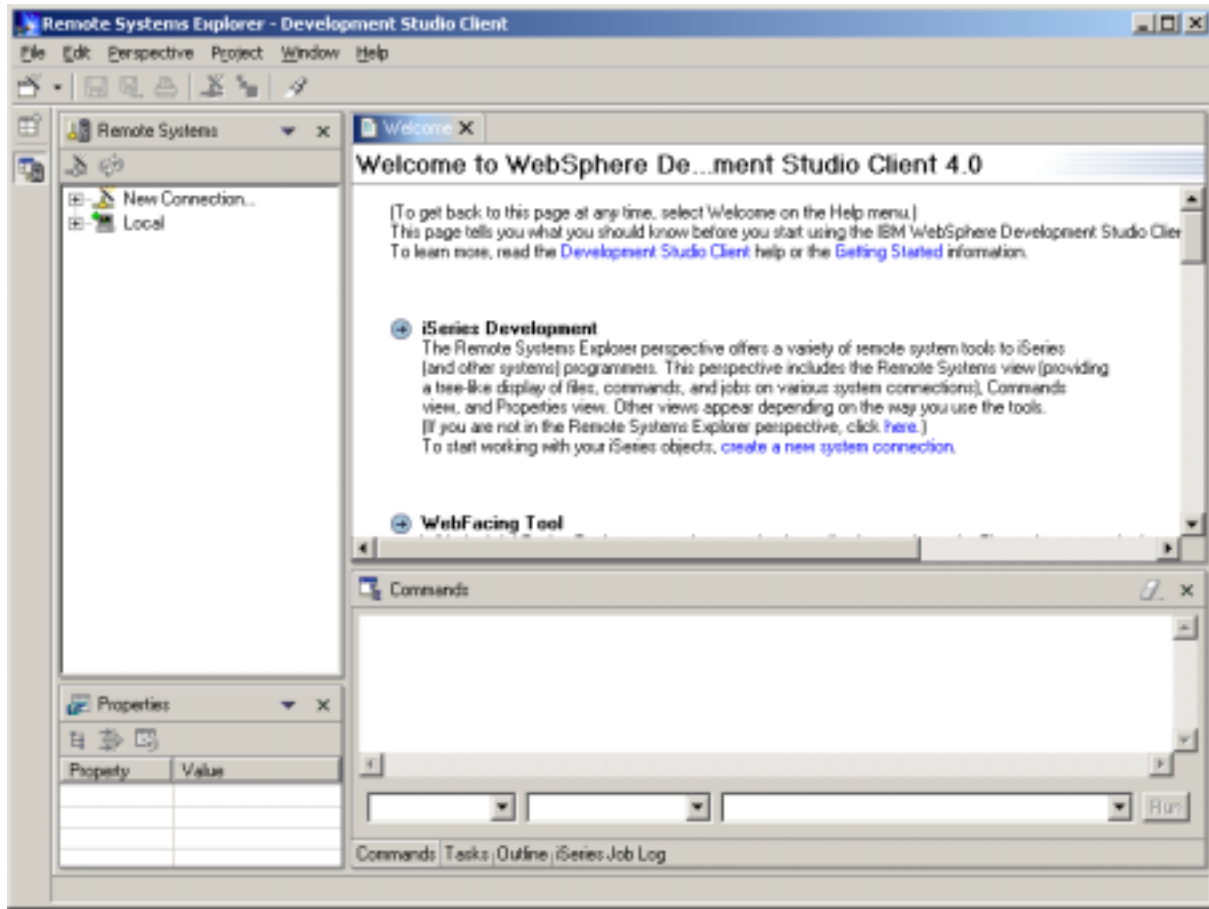
If the workbench has previously been used, it will look different than in the figure above. In case you didn't reset the workbench some pictures will differ from your real lab environment.

You will be working with the **Remote Systems Explorer** (**RSE**) perspective in the workbench.

A perspective is a specific arrangement of views and tools in the workbench, depending on what role the workbench user has, he/she will use a different perspective. A web developer will use the **web perspective**, a Java developer will use the **Java perspective**, an iSeries developer will use the **Remote Systems Explorer perspective**.

The workbench most likely will already show the Remote Systems Explorer perspective, you might see a different perspective already open in the workbench or no perspective.
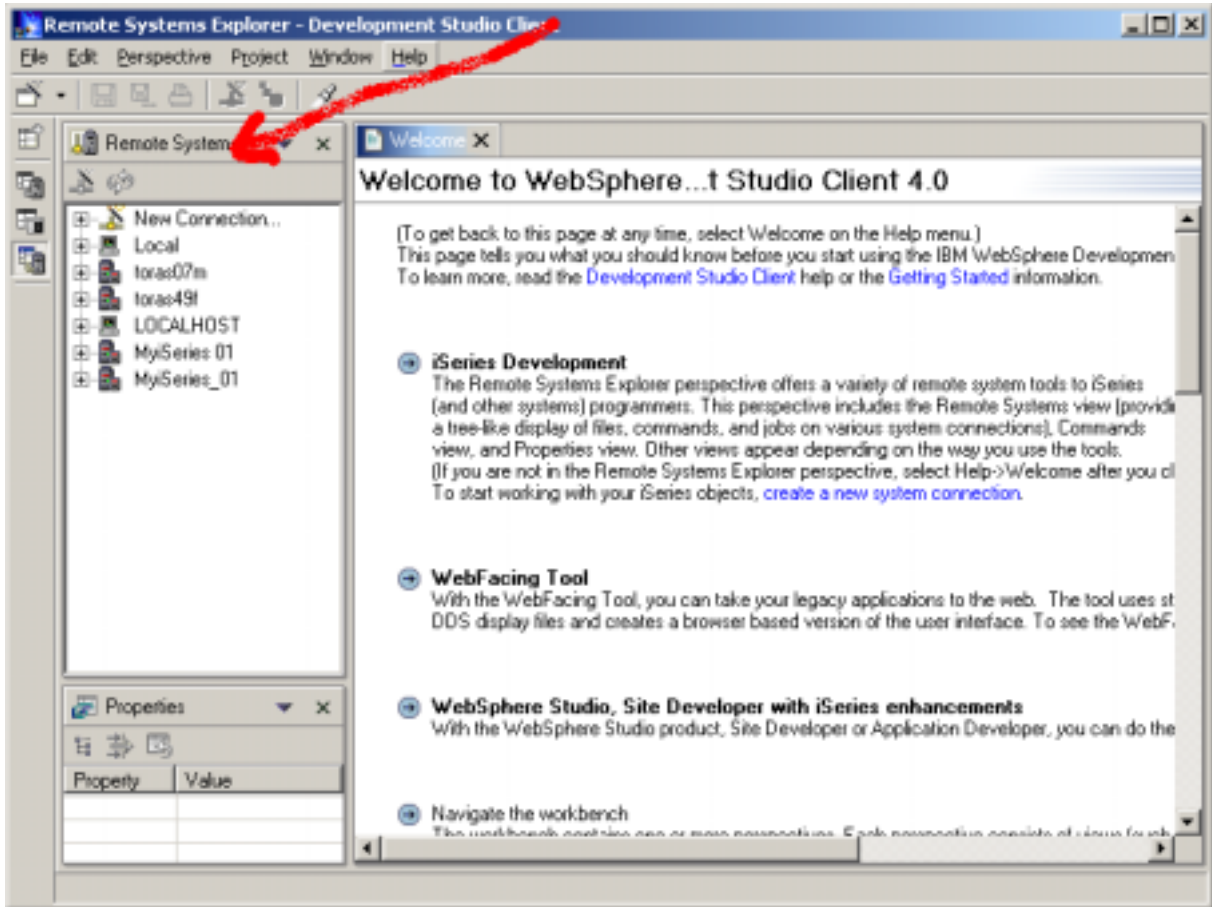
- Check for the name of the perspective, the arrow in the figure above indicates where to look for the perspective name.

    If the active perspective is already the **Remote Systems Explorer** perspective skip the next steps until you see the heading

    Now you are ready to create a connection

Open the Remote Systems Explorer perspective

> **If** there is **no** perspective open or a **different** perspective open, go through the following steps:

- Start the **Remote Systems Explorer** perspective, by selecting the **Perspective** menu item on the workbench **Menu bar**.
- Then Select the **Open** choice from the pull down menu.
- Select the **Remote Systems Explorer (RSE)** option from the sub menu.

Now you are ready to create a **Connection**

- Go thru the following steps to create a connection to the iSeries server we will work with in this Lab.
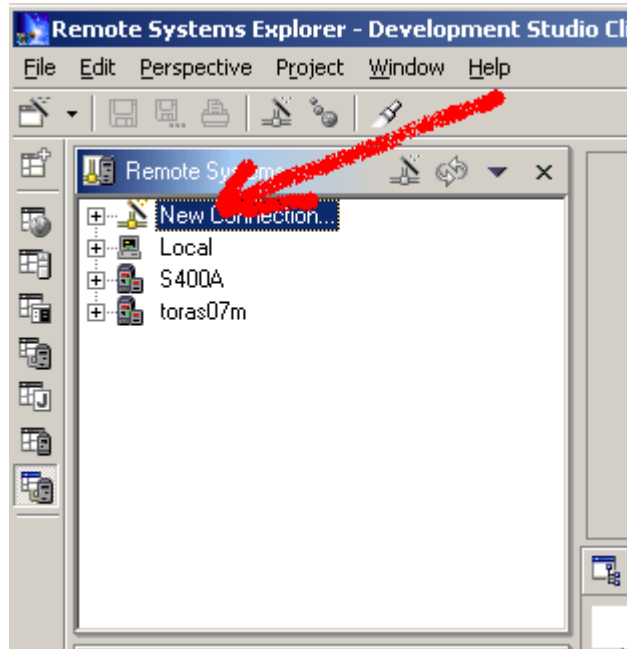
- Locate the **New Connection** node.
- Double click on the **New Connection** node.

The **New Connection** wizard will appear.
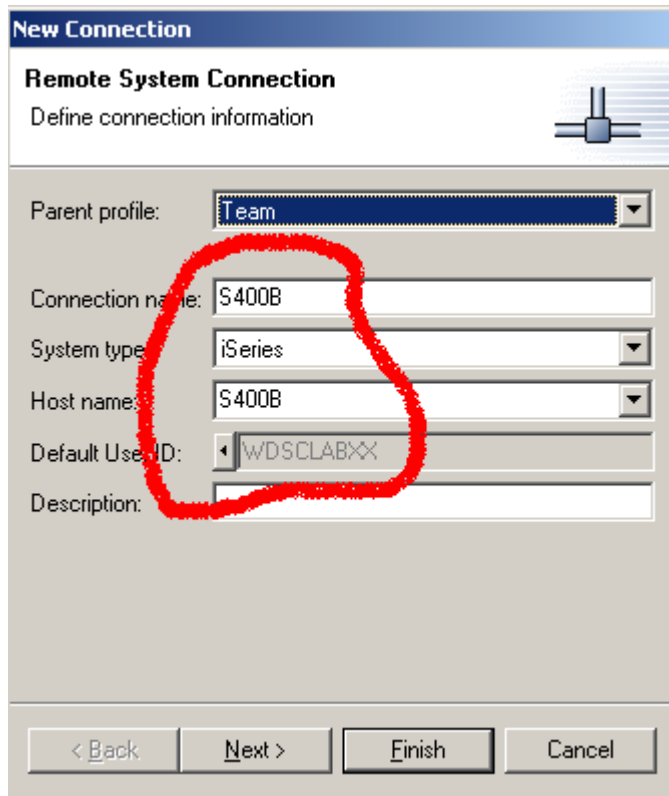
- Leave the parent profile as is, don't change it.

- Specify the **Connection name**, this is the same as your **iSeries host name**.

- Specify the **System type**, which is: **iSeries.**

- Specify the **Host name**, if you don't know it ask your instructor.

- Specify the **Default User ID**, which is **WDSCLABxx**,

   xx being your group number, if you are group/team **70,** your userid would be

     WDSCLAB70

- Press the **Finish** push button.

Selecting an iSeries object in the RSE perspective

Back in the RSE perspective, you now need to get to the iSeries object you want to work with**.**
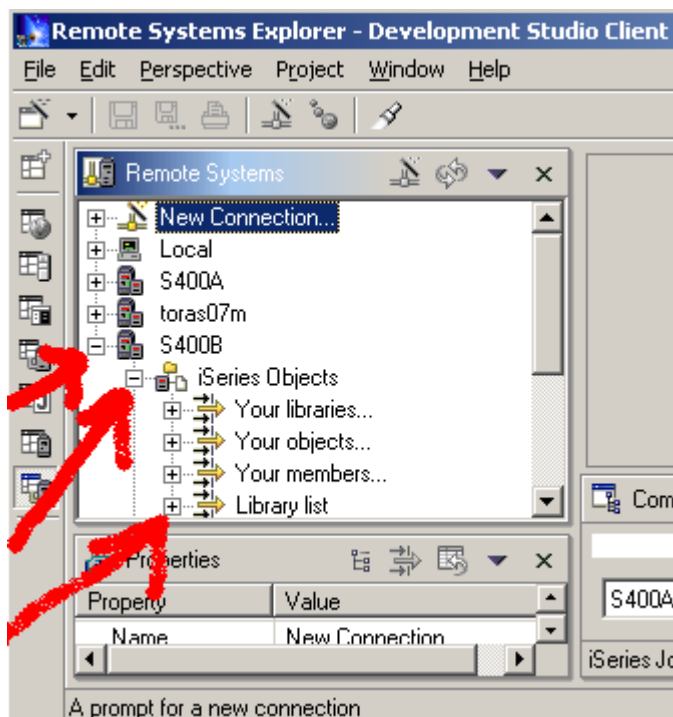
First you will need to specify the library you want to work with:

- Expand the **connection node** that connects to your iSeries host, by clicking on the **+** sign beside it.

- Expand the **iSeries Objects** node.

- Expand the **Library list** node.

We have setup your **Userid** so that your group/team library has been added to the library list automatically.

You will be prompted to sign on to the iSeries server
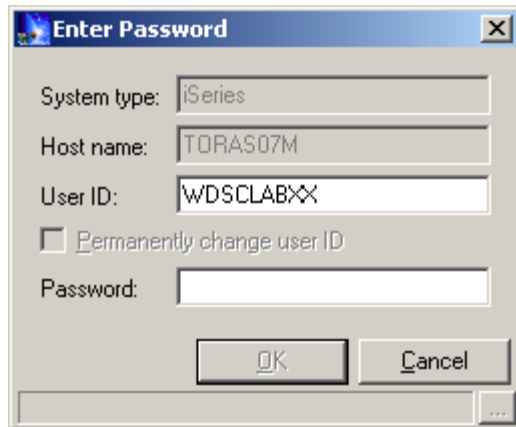
- Use Userid **WDSCLABxx** and password **WDSCLABxx**

Back in the workbench in the RSE perspective you will see the libraries in your job's library list.
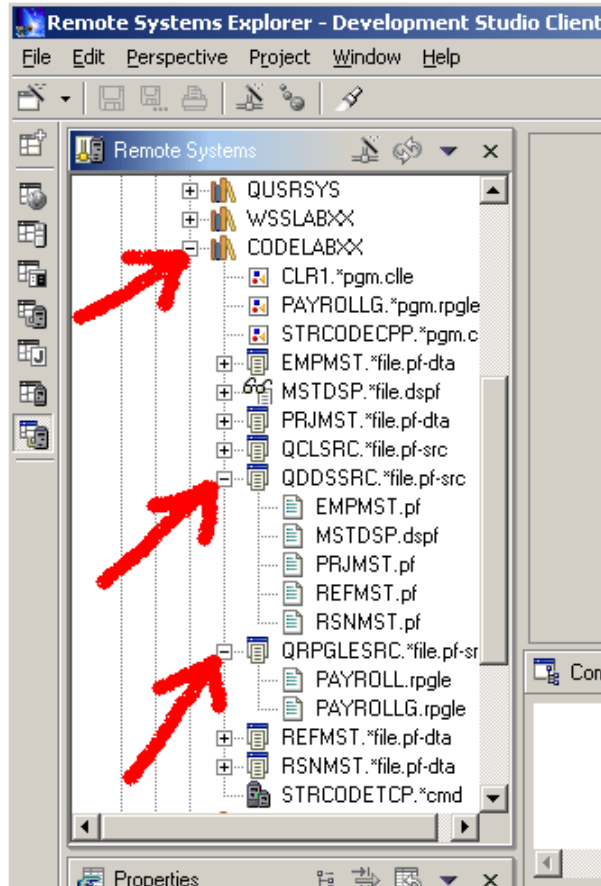
- Locate library **CODELABxx**, **xx** being your group number, and expand it.

Note:

If you see 2 **CODELABxx** library entries in the list, don't worry, one entry represents the **current library**, the other shows the same library, but in the **library list**.

You will see all objects in this library appear in the expanded list.

- Locate the **QDDSSRC** source file and expand it.

- Locate the **QRPGLESRC** source file and expand it as well.

Now you can access the **members** in these 2 source files.
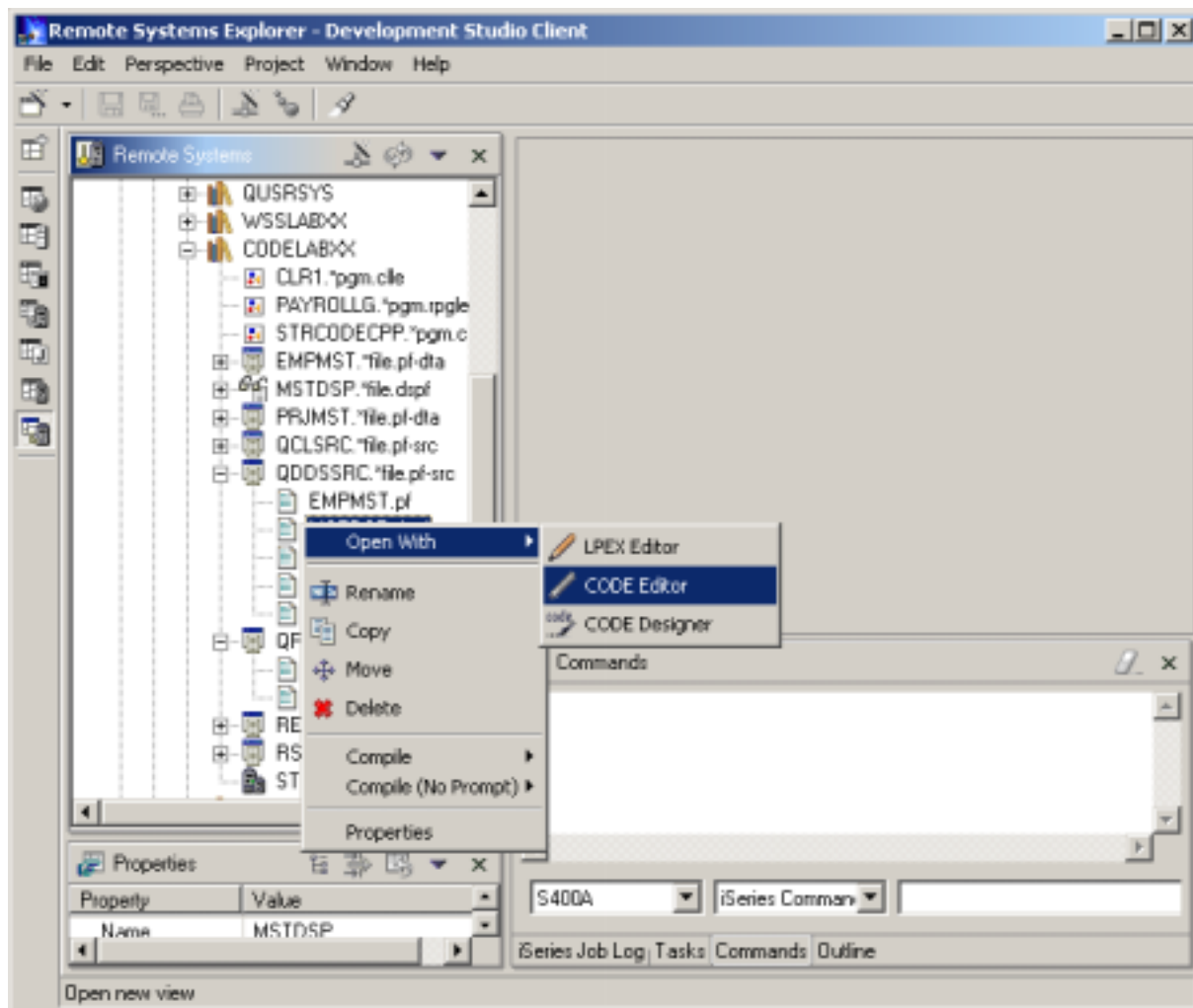
- Right mouse click on member **MSTDSP** in the QDDSSRC source file.

- Select the **Open with** menu option from the pop up menu.

- Select the **CODE Editor** option from the sub menu.

  The CODE editor window will show up with this member loaded.

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 a 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 type of source is in the file. For example, files that have a file extension .RPG are assumed to contain RPG/400 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 RPG/400.

INSTRUCTIONS

2a. The CODE editor should be open already, if you skipped the exercises in Step 1, you can now start the CODE editor from the Windows Start menu. Select **Start→ Programs→ IBM WebSphere Development Tools for iSeries→ IBM CODE400 → CODE Editor**. The CODE Editor appears.

2b. From the editor's '**Options**' menu, select '**Associations**' **->** '**Name pattern**'. The 'Name Pattern Association' dialog comes up.
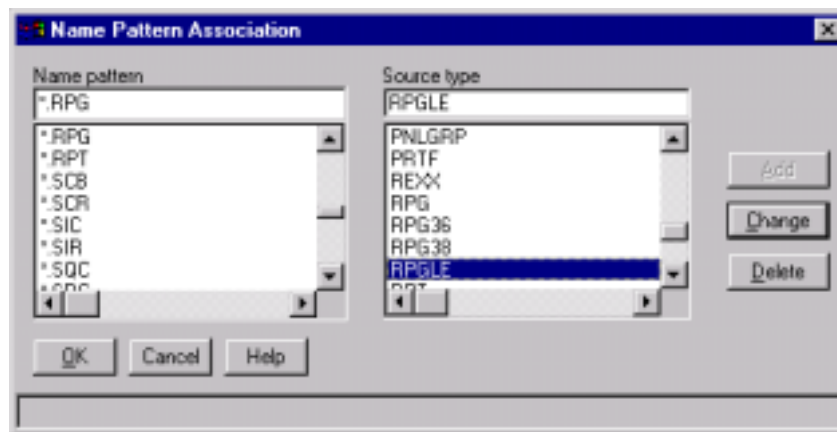
2c. From the 'Name pattern' list box pick the ***.RPG** pattern. Select the **RPGLE** value from the 'Source type' list box.

2d. Press the '**Change**' button to make the changes take effect.
From now on when we open a file with a .RPG extension, the editor will treat it as an **ILE RPG** file.

NOTE: You can associate source types with 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 RPG/400 file.

Now let's get a bit creative. We will invent a new name pattern called 'MySource' and associate it with the CBLLE (which stands for ILE COBOL).

2e. In the 'Name pattern' entry field type: *.**MySource** and then select CBLLE from the 'Source type' list box.
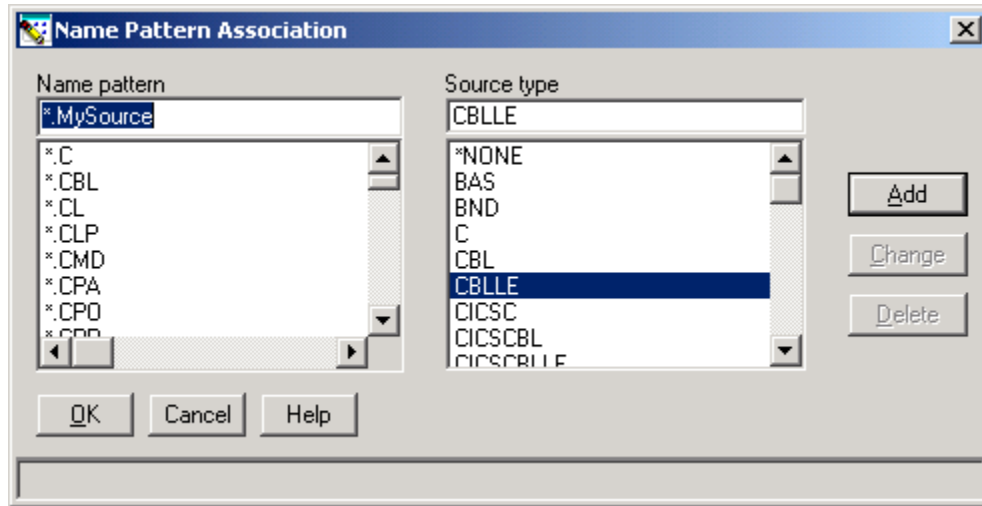
2f. Press the '**Add**' button to complete the association.

2g. Press the '**OK**' button to dismiss the 'Name Pattern Association' dialog.

If we now create a file with the extension .MySource, the editor will treat it as ILE COBOL.

Step 3. Associating source types with language profiles

PURPOSE

In the following exercise you will see the importance of being able to associate name patterns with source types. The CODE editor gives you the flexibility to execute editor commands and macros when a 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 provides you with such a feature!

INSTRUCTIONS

3a. From the editor's '**Options**' menu, select '**Associations**' **-> 'Source types'**. The 'Source Type Association' dialog comes up
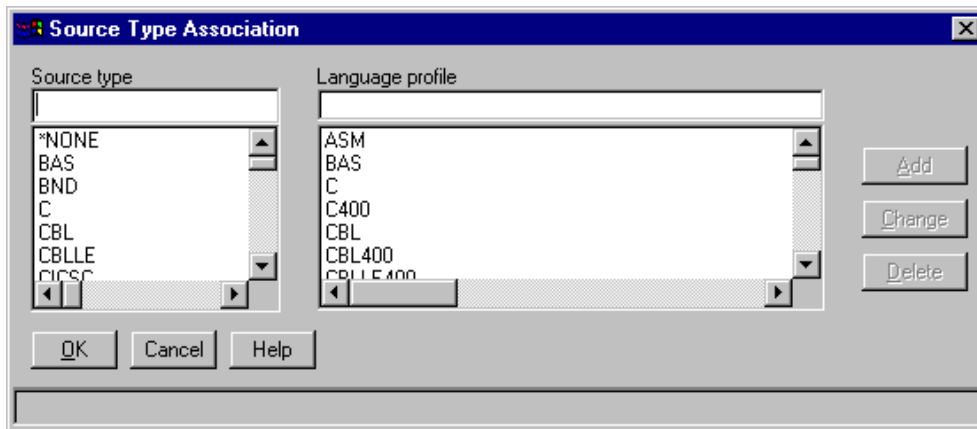
3b. From the 'Source type' list box (on the left) select the **RPGLE** source type. Notice how the **RPGLE400** language profile gets selected in the 'Language profile' list box (on the right).

3c. Press the '**OK**' button to dismiss the 'Source Type Association' dialog

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

Step 4. Executing existing REXX macros

PURPOSE

To get comfortable with running REXX macros from the CODE editor you will now execute two macros that are currently shipped with the WDT/400 product. In order to execute a REXX macro you have to switch to the editor's command line. Use the '**ESC**' key to switch between the source editing area and the command line.
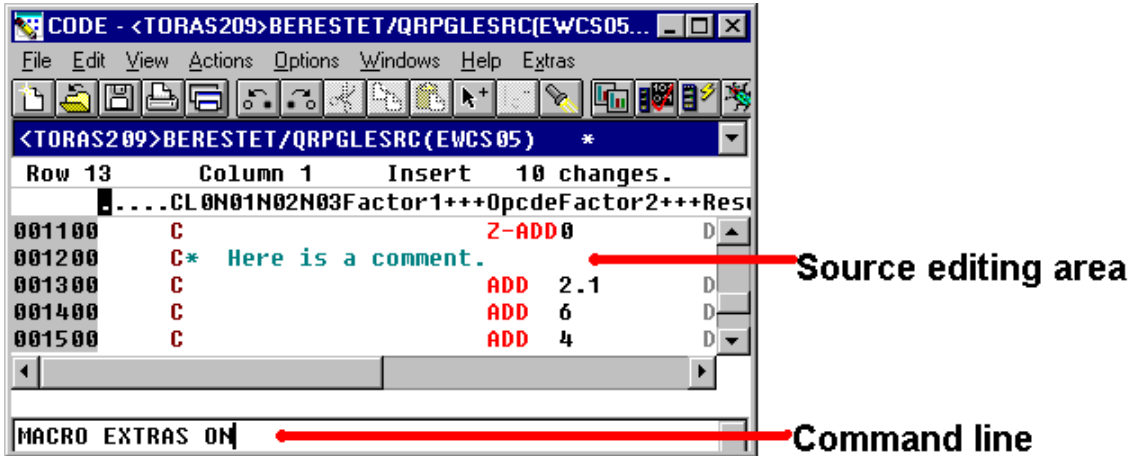
REXX macros are run from the command line by typing: **MACRO MacroName**.
If you are certain that there is no other editor command that matches the name of your macro then the **MACRO** directive can be omitted.

INSTRUCTIONS

Part1: Running a simple REXX macro
4aI. Press the **Esc** key to go to the editor'scommand 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 *product directory*\EXTRAS and then *product directory*\MACROS. 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. Once the editor window gets refreshed, for example after an Open, you will see a new menu item called '**Extras**'.

4cI. Use the RSE to open PAYROLL in the editor. Expand the library **CODELABxx**, expand **QRPGLESRC** and click mouse button two on the member **PAYROLL.** Select **CODE Editor** from the context menu.

Alternatively, you could use the Editor's Open dialog (**File -> Open**, expand **OS400**, expand **CODELABxx**, select **QRPGLESRC** and select **PAYROLL**) or type the following command in the editor command line

<div align="center">

LX <OS400>CODELABxx/QRPGLESRC(PAYROLL)

</div>

where '**xx**' is your workstation number and press the **Enter** key. **LX** is the editor command used to open a file.

```
CODE - <OS400>CODELAB01/QRPGLESRC(PAYROLL)          _ □ ✕
File  Edit  View  Actions  Options  Windows  Help  Extras

<OS400>CODELAB01/QRPGLESRC(PAYROLL)                       ▼
  Row 33        Column 31      Replace
      .....DName++++++++++ETDsFrom+■+To/L+++IDc.Keywords++++++++++++
000031     D*                                                    ▲
000032     D*  Compile time array containing error descriptions.
000033     D ERR          S              50     DIM(10) CTDATA PE
000034     E*
000035     C**************************************************** ▼
◄                                                             ►

|                                                                ▼
1x <os400>codelab01/qrpglesrc(payroll)
Extras on
```
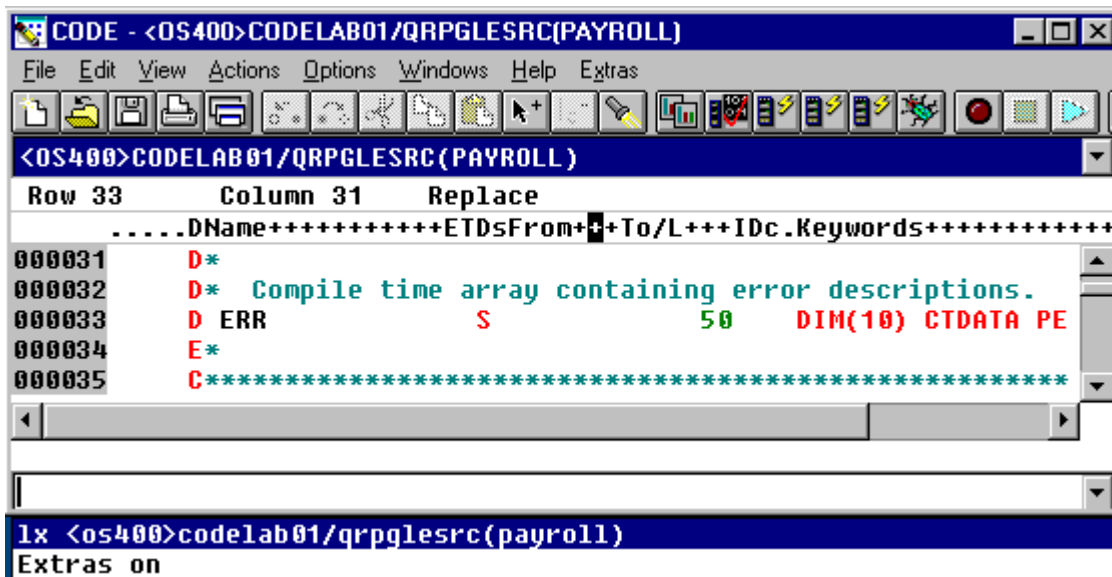
NOTE: Clicking the down arrow in the right hand corner of the editor command line will give you a selection list of the recently-issued editor commands. Just click on a command to recall it and press Enter to re-submit it with or without prior modification.

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 the number of the deleted line.

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 lines are in sequential order again.

4gI. RESEQ is a macro written in REXX. Type:

LX RESEQ.LX

and then press **Enter** to open the macro and see what it does.

```
CODE - H:\PROGRAM FILES\IBM\WDT400\MACROS\reseq.lx

File  Edit  View  Actions  Options  Windows  Help  Extras

H:\PROGRAM FILES\IBM\WDT400\MACROS\reseq.lx

 Row 1          Column 1       Insert
          ---+----1----+----2----+----3----+----4----+----5----+----6----
00001/* RESEQ INCR START */
00002
00003'EXTRACT PREFIXFORMAT INTO FORMAT'
00004if FORMAT <> '999999XXXXXX' then do
00005    'EXTRACT NAME'
00006    say 'RESEQ.LX: File' NAME 'does not have sequence numbers'
00007    exit
00008    end
00009
00010parse arg INCR START .
00011
00012/* if the parms aren't specified, check global variables */
00013if INCR = '' then
00014    'EXTRACT GLOBAL.RESEQOPT_INCR INTO INCR'
00015if START = '' then
00016    'EXTRACT GLOBAL.RESEQOPT_START INTO START'
00017
00018/* now use the same defaults as LPEX */
00019if INCR = '' then
00020    INCR = 100
00021if START = '' then
00022    START = INCR
00023
00024'PREFIXRENUMBER' INCR START
```

This is what the macro RESEQ looks like. It may seem a little cryptic now, but once we take a closer look, macros will not seem mysterious any more.

Switching between files:

Multiple files can be loaded into the CODE editor simultaneously. In order to switch from one file to another, there is a drop-down list which is located directly under the toolbar. When you click on the down arrow on the right, the entire list shows up and you can select the file from there.
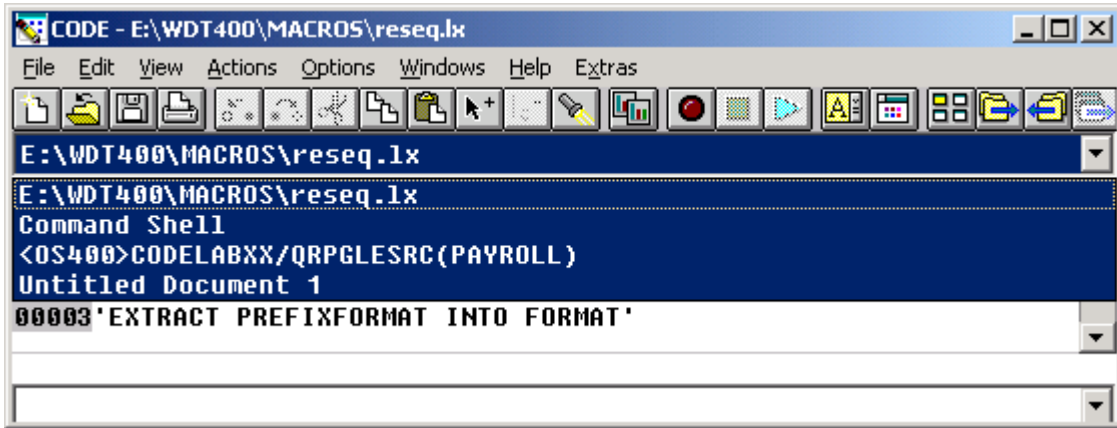
Part 2: Running a REXX macro with prompt

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. Play with the options that are available from the '**Extras**' menu. You can get more information about the supplied 'extra features' by exploring the 'CODE/400Tips and Techniques' available from the '**Extras**' **->** '**Information**' menu.

4bII. Press the **Esc** key to go to the command line if you are not already there.

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

Course material may not be reproduced in whole or in part
without the prior written permission of IBM.
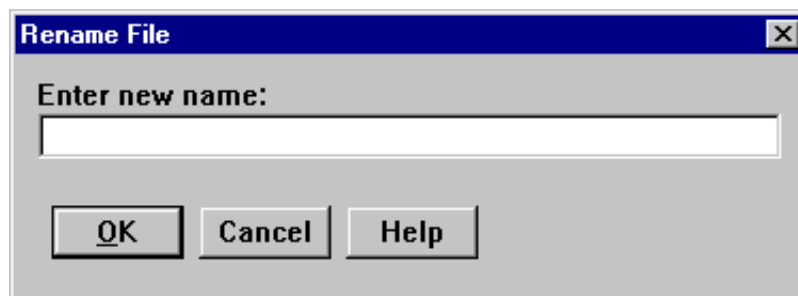
4dII. Enter **RENAMED.DAT** in the 'Rename File' entry field for the new file name and then press the '**OK**' button.

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

```
CODE - RENAMED.DAT                                          _ □ ✕
 File  Edit  View  Actions  Options  Windows  Help

RENAMED.DAT                                                       ▼
 Row 1          Column 1        Insert
      ■---+----1----+----2----+----3----+----4----+----5----+----6----+--
00001/* RESEQ INCR START */                                      ▲
00002
00003'EXTRACT PREFIXFORMAT INTO FORMAT'
00004if FORMAT <> '999999XXXXXX' then do
00005    'EXTRACT NAME'
00006    say 'RESEQ.LX: File' NAME 'does not have sequence numbers'
00007    exit
00008    end
00009
00010parse arg INCR START .
00011
00012/* if the parms aren't specified, check global variables */
00013if INCR = '' then
00014    'EXTRACT GLOBAL.RESEQOPT_INCR INTO INCR'        ▼
```
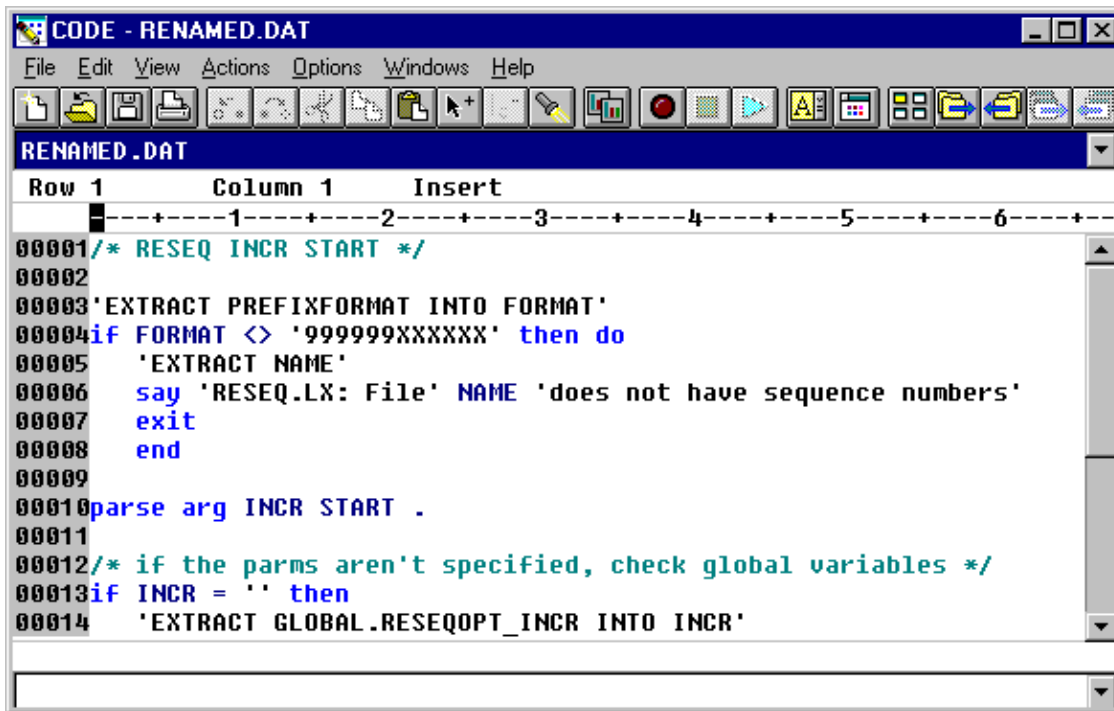
4fII. As you might have suspected already, **RENAME** is another REXX macro. Type:
LX RENAME.LX
and then press the **Enter** key to bring up its source in the editor.

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 mean:
1) Set the dialog title to "Rename File"
2) Create a dialog label called "Enter new name:"
3) Read up to 255 characters from the entry field.
You will use similar code in the following exercises when the 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 if you are not already there.

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 - H:\PROGRAM FILES\IBM\WDT400\MACROS\rpgproc.lx       _ □ ×

File  Edit  View  Actions  Options  Windows  Help  Extras

H:\PROGRAM FILES\IBM\WDT400\MACROS\rpgproc.lx             ▼

Row 1         Column 1        Insert
     █---+----1----+----2----+----3----+----4----+----5----+----6----
00001/*******************************************************/    ▲
00002/* RPGPROC.LX                                      */
00003/*                                                 */
00004/* This macro builds an RPG procedure template     */
00005/*                                                 */
00006/*******************************************************/    ▼
◄                                                            ►
H:\PROGRAM FILES\IBM\WDT400\MACROS\rpgproc.lx saved

                                                             ▼
```

5d. At this point you should save the file. Use the **'File'->'Save as...'** command and add the file to the WDT/400 macros directory:
> **x:\WDSC\MACROS\RPGPROC.LX**

(x is the drive where CODE is installed).
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** and press **Enter**. Nothing happens, the macro does not do anything yet.

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:

<p style="color:teal; text-align:center">**/</p>

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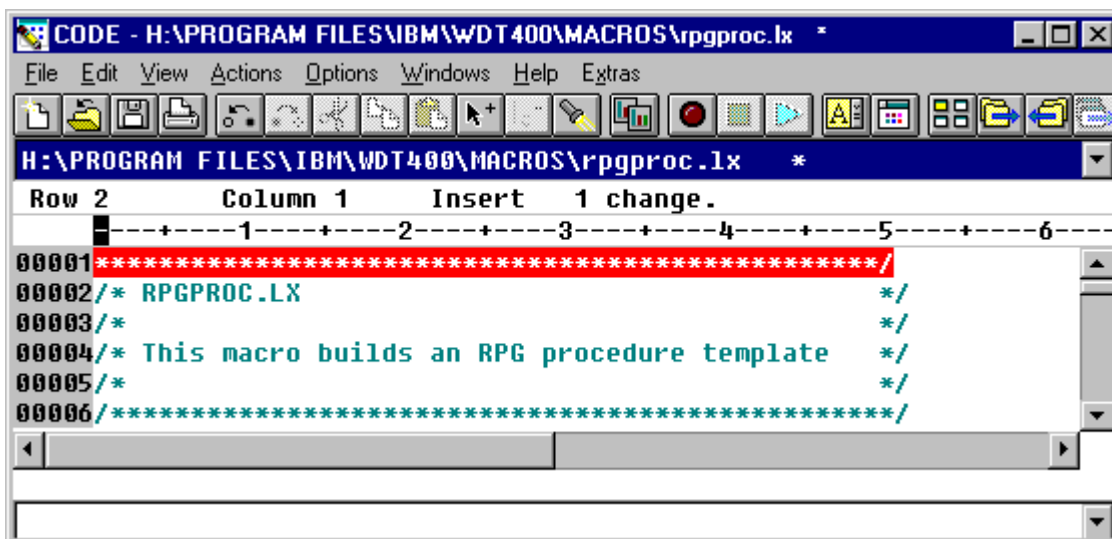
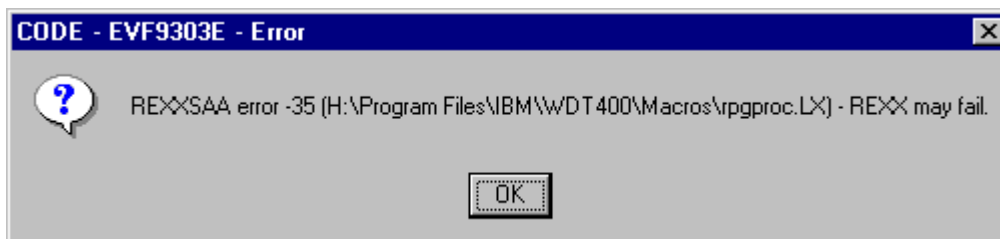
5g. Save the file - this time use the 'Save' icon on the toolbar.
It looks like this:

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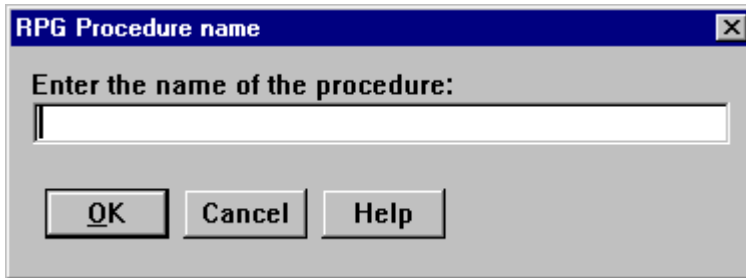

NOTE: If you want more information about an error when running a macro, select Macro Log

---

from the editor's Windows menu.

5h. Correct the error by putting the '/' character back 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 the following

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 editor's '**Help**' menu select the '**Editor reference'** option. The online *Editor Reference* manual comes up in a browser and displays the navigation page *Editor Commands and Parameters*. We are interested in information about the command and parameter *lineread*. Click on '**Commands Summary'** and page down. Click on the '**lineread**' command to display the description and carefully read the documentation and example.
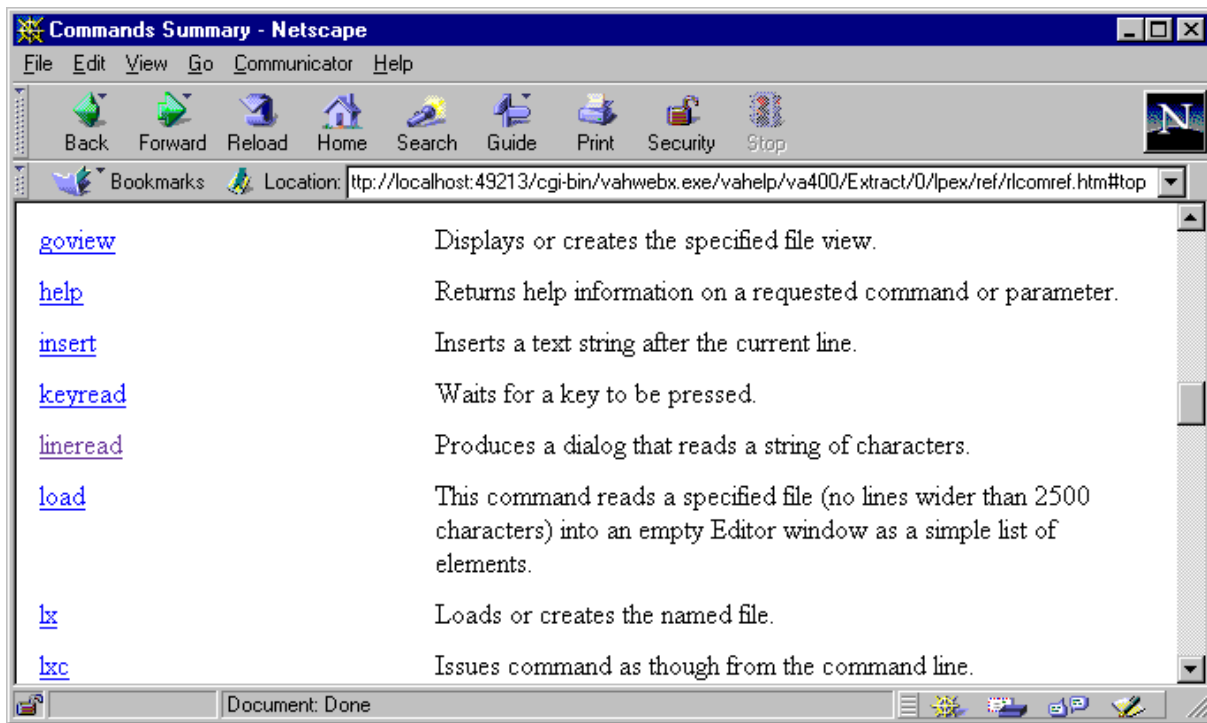
5j. Use the Back button in the browser's tool bar to get back to the command selection. Page all the way up and press the Synchronize button in the top right corner. An index and a search entry field appear. Type *'lineread '* in the search field and press Enter. Select the entry '**Editor - lineread parameter**. Read the documentation and examples. Minimize the help window, we will need it again later.

The following lines of REXX code will set up the dialog box title, a prompt label, and an entry field of length 10:

```
'set lineread.title RPG Procedure name'
'set lineread.prompt Enter the name of the procedure: '
'lineread 10 '
```

5k. Now that we understand how to show a dialog box, we still need to figure out how to read the procedure name that the user has entered, and which button, **OK** or **Cancel,** was pressed. We will not worry about the '**Help**' button. 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 canceled'
   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 editor's '**Help**' menu select the '**REXX help'** option. You will find the '**Programming guide'** and '**Reference**' manuals.

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 the 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 the procedure name entered by the user with blanks (to exactly 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 this:

```
/* 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'
```

Note: Since the macro will later insert these lines into an RPG source file, the spacing should be exactly as shown to match the RPG columns. There are 6 blanks between *insert* and the specification entry.

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

```
CODE - H:\adtswin\macros\rpgproc.lx  *                                    _ □ ×
File  Edit  View  Actions  Options  Windows  Help  Extras

H:\adtswin\macros\rpgproc.lx      *
 Row 7         Column 1      Insert    2 changes.
      ----+----1----+----2----+----3----+----4----+----5----+----6----+----7--
00001/****************************************************/
00002/* RPGPROC.LX                                      */
00003/*                                                 */
00004/* This macro builds an RPG procedure template     */
00005/*                                                 */
00006/****************************************************/
00007'set lineread.title RPG Procedure name'              /* Set dia
00008'set lineread.prompt Enter the name of the procedure: '   /* Prompt
00009'lineread 10 '                                        /* Create
00010'extract lastline'                                    /* Read te
00011'extract lastkey'                                     /* What ke
00012
00013if ((lastline = '') | (lastkey = 'ESC')) then do
00014   'msg Request cancelled'
00015   exit
00016end
00017
00018procName = lastline
00019/* Pad procName with blanks to make it 10 characters long */
00020do procLength = length(lastline) to 9
00021   procName = procName' '
00022   end
00023/* The procName is 10 characters long including blanks */
00024'insert       D* ----------------------------------------'
00025'insert       D* Prototype for procedure: 'procName
00026'insert       D* ----------------------------------------'
00027'insert       D 'procName'      PR'
00028'insert                                    .'
00029'insert       P* ----------------------------------------'
00030'insert       P* Procedure Name: 'procName
00031'insert       P* Purpose: '
00032'insert       P* ----------------------------------------'
00033'insert       P 'procName'      B'
00034'insert       D 'procName'      PI'
00035'insert                                    .'
00036'insert       C* Your calculation code goes here'
00037'insert                                    .'
00038'insert       C                        RETURN'
00039'insert       P 'procName'        E'
```

40

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** in 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 the editor views the TESTFILE.RPG as an ILE RPG file (the default is RPG/400)!

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 an ILE RPG file, bring up the **'File Properties'** dialog from the '**File**' -> '**Properties...**' editor menu.
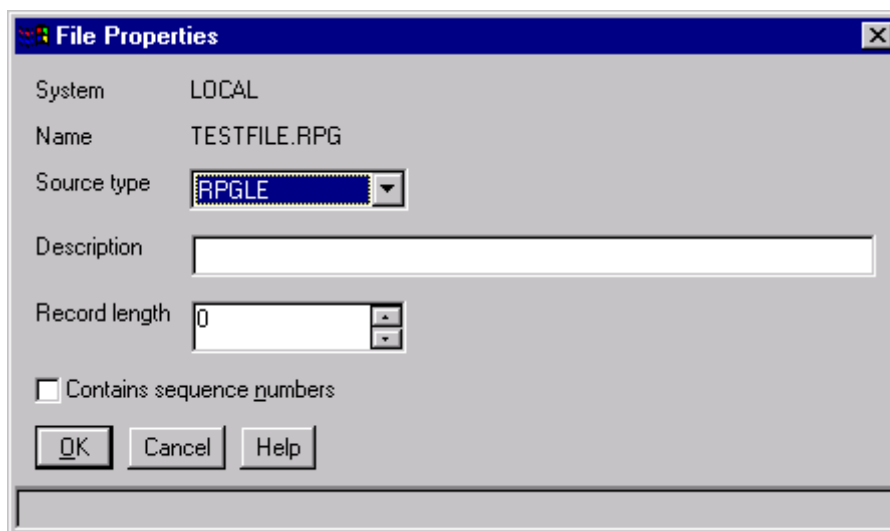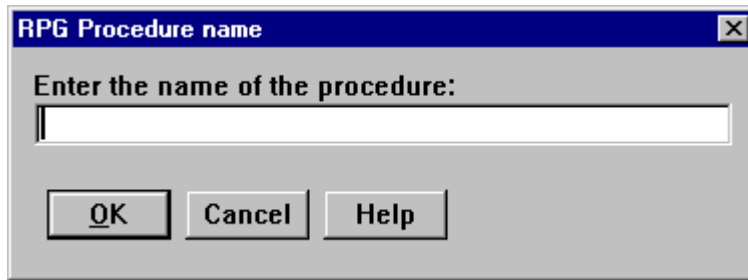

Notice that the field 'Source type' contains the value RPGLE. This means that the currently loaded file is an ILE RPG file. If necessary the value could be changed right here.
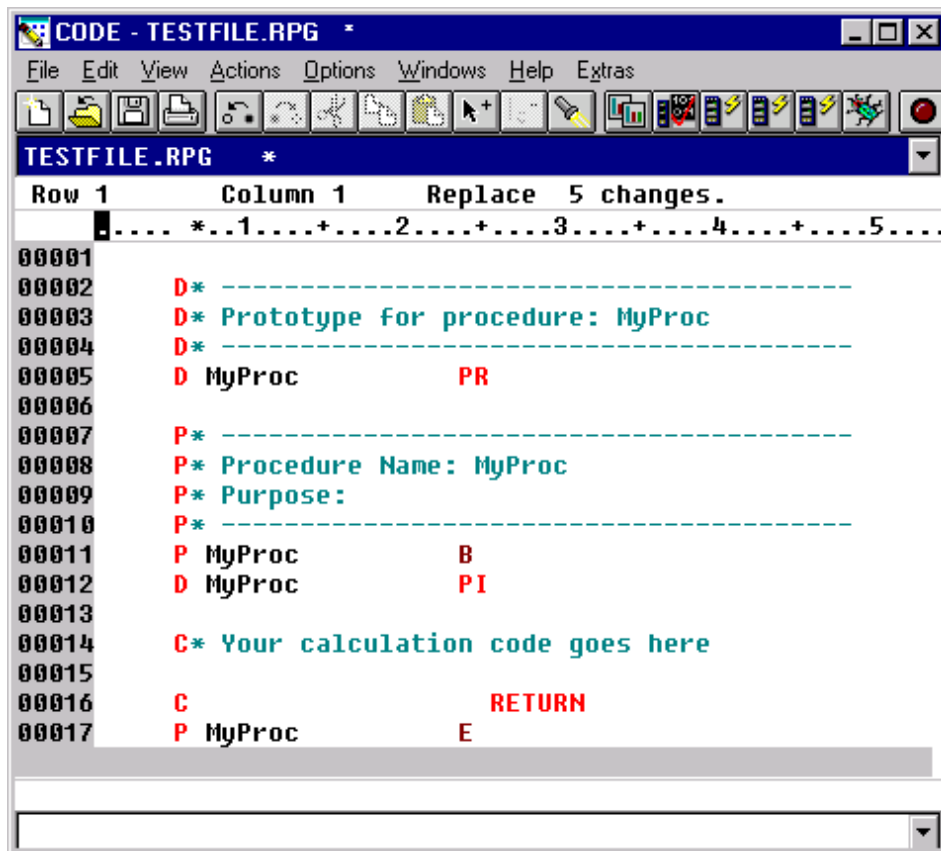
5q. Click the 'Cancel' button to dismiss the dialog.

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:

```
┌─────────────────────────────────────────────────┐
│ RPG Procedure name                           [X] │
├─────────────────────────────────────────────────┤
│                                                  │
│  Enter the name of the procedure:                │
│  ┌────────────────────────────────────────────┐  │
│  │                                            │  │
│  └────────────────────────────────────────────┘  │
│                                                  │
│    ┌──────────┐  ┌──────────┐  ┌──────────┐     │
│    │    OK    │  │  Cancel  │  │   Help   │     │
│    └──────────┘  └──────────┘  └──────────┘     │
│                                                  │
└─────────────────────────────────────────────────┘
```

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

```
┌─────────────────────────────────────────────────────────┐
│ 🔷 CODE - TESTFILE.RPG  *                      [_][□][X] │
├─────────────────────────────────────────────────────────┤
│ File  Edit  View  Actions  Options  Windows  Help  Extras│
│ [toolbar icons]                                          │
├─────────────────────────────────────────────────────────┤
│ TESTFILE.RPG    *                                     [▼]│
│ 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                          │
│ 00006                                                    │
│ 00007     P* ---------------------------------------     │
│ 00008     P* Procedure Name: MyProc                      │
│ 00009     P* Purpose:                                    │
│ 00010     P* ---------------------------------------     │
│ 00011     P MyProc           B                           │
│ 00012     D MyProc           PI                          │
│ 00013                                                    │
│ 00014     C* Your calculation code goes here             │
│ 00015                                                    │
│ 00016     C                  RETURN                      │
│ 00017     P MyProc           E                           │
│                                                       [▼]│
└─────────────────────────────────────────────────────────┘
```

Note: If any of the lines are marked by an error message, your template is causing a syntax error. Most likely the columns are misaligned. Correct the error and move the cursor off that line to get the syntax checked again. Don't forget to change the corresponding line in the macro!

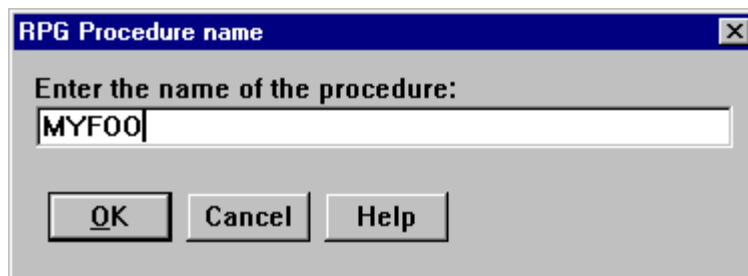# Optional exercise - prefilling the procedure name entry field

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 5r), the 'Procedure Name' entry field is empty. Sometimes it is useful to prefill an entry field with some default value.

INSTRUCTIONS

Modify your REXX macro so that the 'Procedure Name' entry field contains the value
                                MYFOO
when the prompt dialog comes up.

```
RPG Procedure name                                    ×

Enter the name of the procedure:
MYFOO|

    OK        Cancel      Help
```

HINT

Read the **lineread** editor command in the 'Editor Reference' manual.

Step 6. Updating the editor's menu bar

PURPOSE

Once the REXX macro is written you can invoke it from the editor command line. However, for frequent use this may become tedious. In such cases, we can use the editor commands to create new menu items. One of the command's parameters 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 *'Parameters Summary'* of the *'Editor Reference'* manual and get familiar with this command.
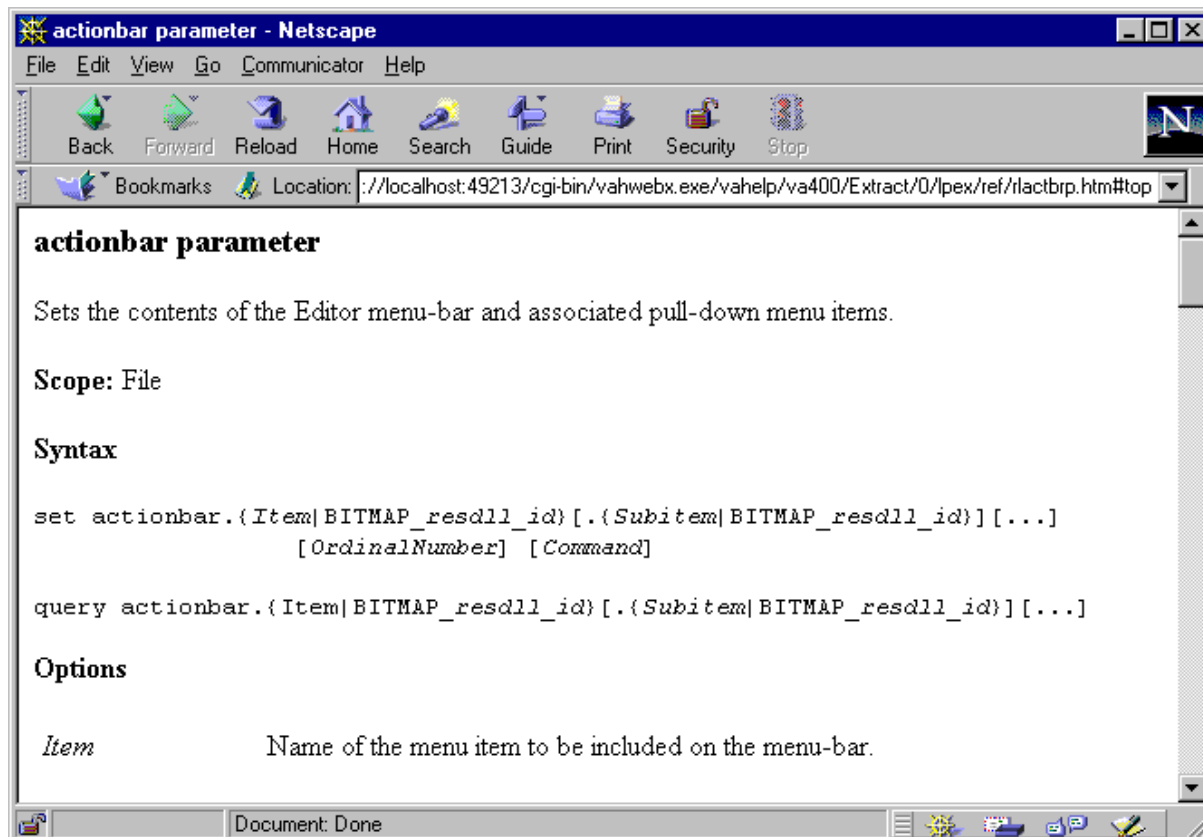
6b. Switch to the editor command line and type the following command:
SET ACTIONBAR.E~xtras.~COMMON.RPG~PROC\tCtrl+Z MACRO RPGPROC
and press **Enter**.

The resulting menu item will be:
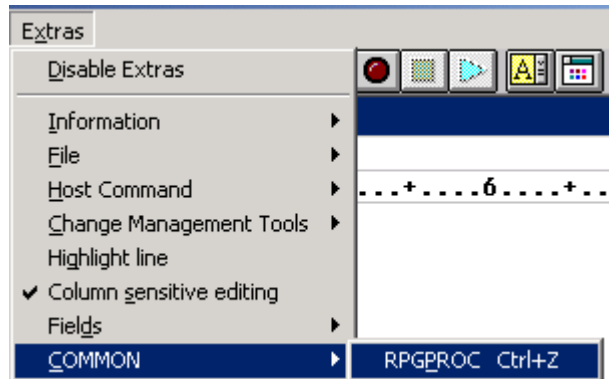
COOL!

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 expect them to!

Step 7. Updating the editor's 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 commands that allow you to update the editor's toolbar and popup menu with the items of 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 our famous RPGPROC macro.

INSTRUCTIONS

7a. Use the **TOOLBAR** editor command to add a button to the CODE editor toolbar.
 Browse the *'Editor Reference'* manual 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:
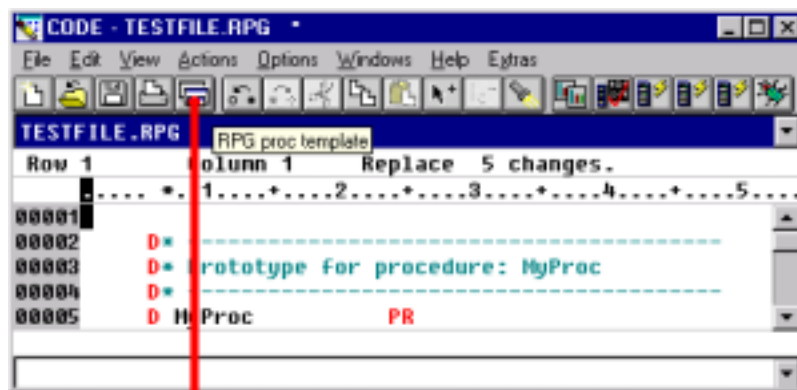
New Toolbar Button

Notice that in this example you used the value **_33** for the **BITMAP** option. Bitmaps shipped by CODE 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.

Popup Menu: An example of 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.

7c. Use the **POPUPMENU** editor command to add items to the CODE editor popup menu. Browse the *'Editor Reference'* manual to learn about this command.

7d. 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 up the popup menu the item RPG<u>P</u>ROC is added:
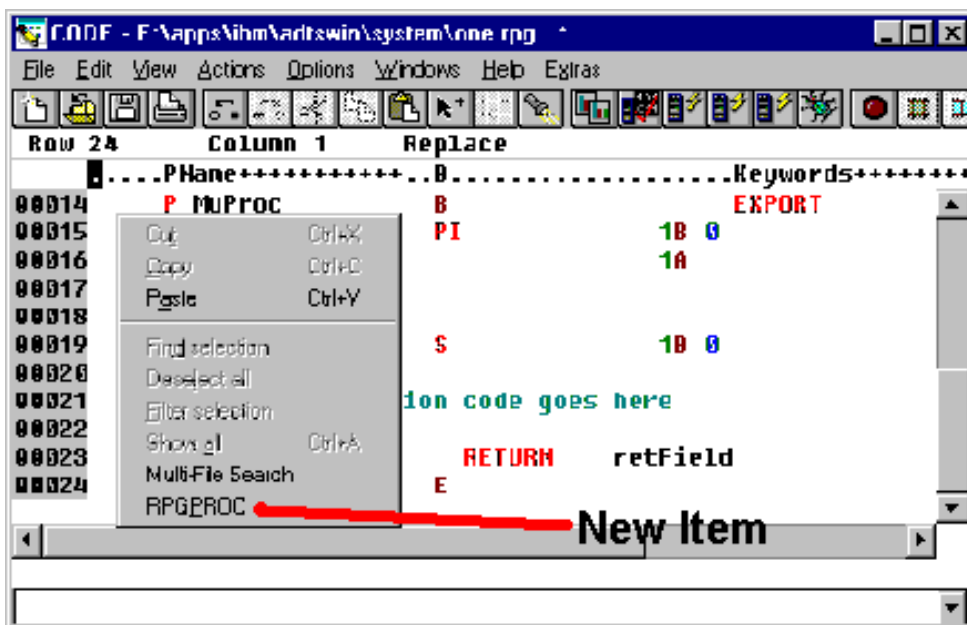
7e At this point you can play with the newly created toolbar button and popup menu item. Make sure they both behave the way you expect them to! **Cool stuff!**

Step 8. CODESRV - remote execution command

PURPOSE
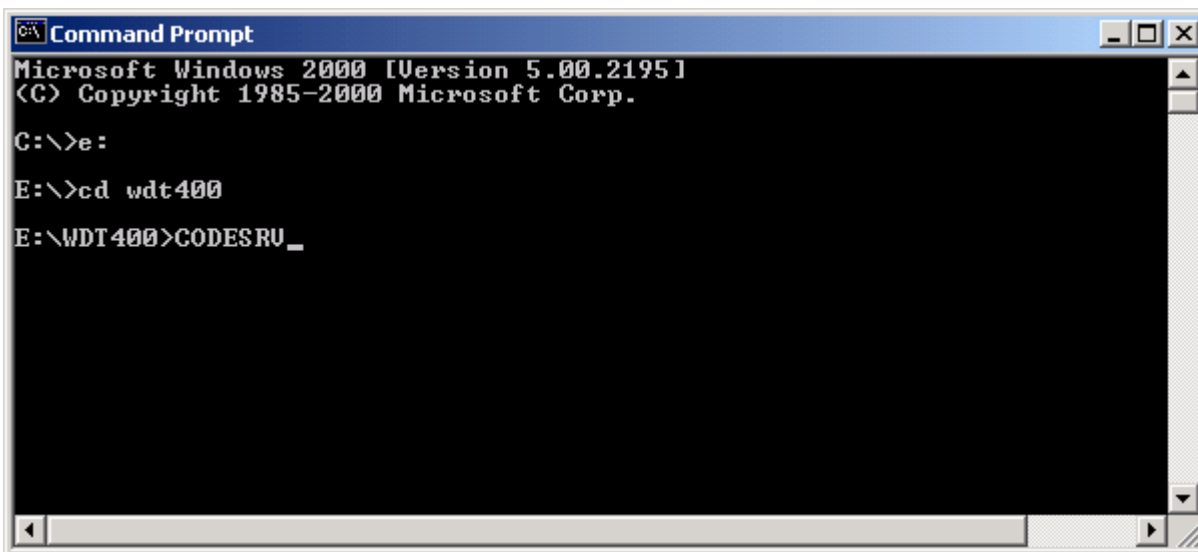
The CODESRV command is a workstation command that can be used to:
- Get a list of the active host CODE servers
- Send commands to the iSeries
- Download and upload source
- Get lists of objects that match a specified filter.

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 the CODESRV command to become really useful, we must make sure that the CODE communication server is started (see **Step 0**).

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

```
Command Prompt                                            _ □ ×
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>e:

E:\>cd wdt400

E:\WDT400>CODESRV_
```

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

INSTRUCTIONS

8a. In the MS-DOS prompt ensure that you are in directory x:\WDSC (x is the drive where WDSc is installed). To see the list of active CODE servers type:

       CODESRV SERVER

and then press **Enter**. Your list should have **OS400** in it; there may be additional entries.

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

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

A spool file of MSTDSP is created on the iSeries.

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 help for the command.

If you are really ambitious, use **CODESRV GET OS400...** and **CODESRV PUT OS400...** to download and upload members to the iSeries. Notice in the help that you can also use the CODESRV command to shut down all servers (you can have up to twenty connections at a time) or the connection to a specific server.

NOTE: You can also invoke CODE tools from the iSeries. 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 QDEVTOOLS/EVFCFDBK PARM('37' 'Y' 'OS400' '<LOCAL> CODEEDIT
                               "<server>lib/file(member)"')

If your iSeries is running V4R5, use:

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

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 the current member being edited on the host.

8e. 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 is used to submit the SEU print option (OPTION (6)) to the iSeries host.

Step 9. CODE editor profiles

PURPOSE

The menu items, toolbar buttons, and shortcuts that you created in the 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 the file **RPGLE400.LXL** into the editor. Alternatively, you could use the open dialog by selecting **File -> Open...** expand **Local**, expand the drive where WDSc is installed, expand **WDSC**, click on **macros**, scroll down to **rpgle400.lxl** and double click on it.

9b. Look through the file. It contains various editor commands that run when an 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 parser type ILEPAR...

```
'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 create 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.

```
CODE - H:\PROGRAM FILES\IBM\WDT400\MACROS\rpgle400.lxu
File  Edit  View  Actions  Options  Windows  Help  Extras

H:\PROGRAM FILES\IBM\WDT400\MACROS\rpgle400.lxu
 Row 1          Column 1       Insert
      ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+
00001/*****************************************************/
00002/* RPGLE400.LXU - Language-sensitive, user-defined */
00003/*               editor profile for ILE RPG        */
00004/*****************************************************/
00005
00006/* Setup the menu, toolbar, and pop-up menu items   */
00007'SET ACTIONBAR.E~xtras.~RPGTools.RPG~ProcTemplate\tCtrl+Z MACRO RPGPROC'
00008'SET TOOLBAR.RPGProc BITMAP _33 HELP "RPG proc template" 4 MACRO RPGPROC'
00009'SET POPUPMENU.RPG~Proc MACRO RPGPROC'
```

9e. Save the file in directory
> **x:\WDSC\MACROS**

Close the editor 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 the editor and creates a new file COMMON.RPG.

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 the editor loaded the ILE RPG file! Remember that in step 2 of the exercises we associated the name pattern *.RPG with the source type RPGLE and that the source type RPGLE is associated with the RPGLE400 language profile.

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

9g. Close the CODE editor.
<div align="center">

This section of the lab is complete!

</div>

<u>The Lab - Section 2: Lpexlets</u>

<u>Section Introduction</u>

In this section we will learn how to program the CODE 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 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, you use the RUNJAVA Lpexlet_Class_Name command.

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

- As a language that helps you customize the CODE editor via Lpexlets.
- As a programming language for your client user interfaces.
- (Since V4R2), as a programming language on the iSeries.

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 iSeries 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 **iSeries Toolbox for Java** set of classes written by IBM Rochester. This Java code offers a significantly easier means to access iSeries 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 **iSeries Toolbox for Java** code can be used for Java applications or applets; The **iSeries Toolbox for Java** classes are shipped with WDT/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*.

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 which implements a Java-supplied interface for handling GUI events.
- Compile Java classes using the CODE 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 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 on our journey to CODE Lpexlets...

Step 1. Creating an RPGProc Lpexlet Class

Java Reference:

- **Comments** in Java come in two forms:
    - **Multiple lines**: 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 iSeries 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 iSeries 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 **modules**! Modules contain variables and RPG procedures and subroutines. Java classes contain variables and methods. ***Methods*** are like RPG ***procedures***
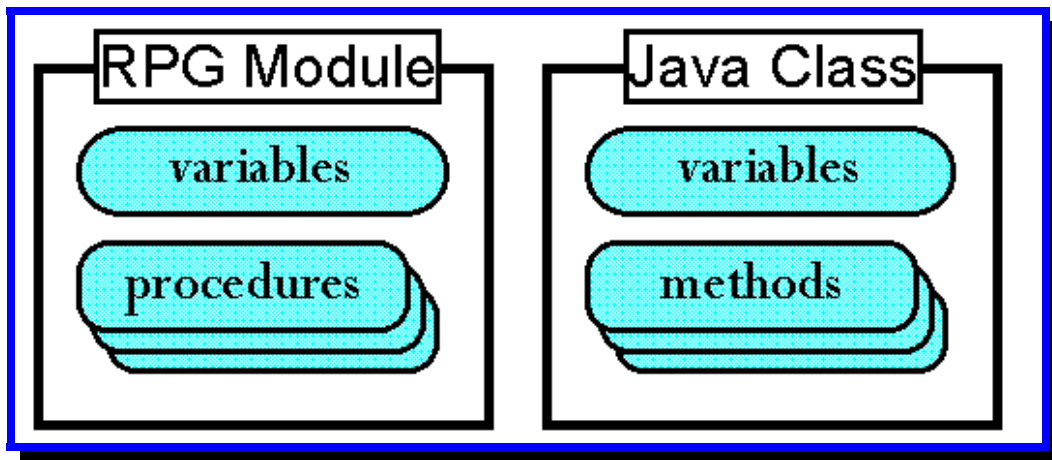
A class in Java typically looks like this:

```
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 than the inherited one. At run time Java decides which method to use appropriately. This feature of the Java language is called **polymorphism**.

END Java Reference

58

PURPOSE

CODE 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 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 the **'**RUNJAVA Lpexlet_Class_Name' command is run.

In our case Lpexlet_Class_Name will be RPGProc and hence the command becomes 'RUNJAVA RPGProc'. Don't run anything yet!

The RPGProc Lpexlet will put up a nice dialog prompting the user for the Procedure Name and the Programmer Name.
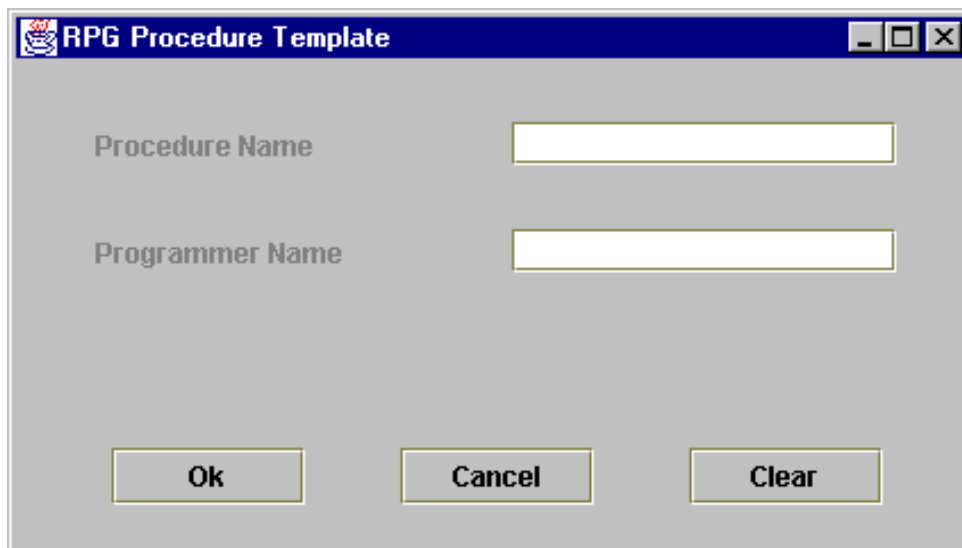
Once all information is entered, the Lpexlet will call a REXX macro to generate the procedure template. The reason for this is very simple - we already have code that does this job. So we will reuse a part of the RPGPROC macro.

INSTRUCTIONS

1a. Start up the CODE editor and open the file RPGProc.java. Remember that in Java file names are case sensitive!

> **x:\CODELAB\RPGProc.java**

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

```
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 corresponds to the name of the source file (not counting the **.java** extension). 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 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 in 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 the *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 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.

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.

1c. Take a close look at the code of RPGProc.java. Pay special attention to the statements that set the editor variables PROCNAME and PGMRNAME.

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

> **SAVE "x:\WDSC\JAVA\RPGProc.java"**

and then pressing **Enter**.

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
The RPGProcFrame class will inherit from the class **JFrame**. JFrame is a Java-supplied class. It is responsible for putting up the dialog window and border. Other Java-supplied classes that are used by the RPGProcFrame class are:
* **JPane**. The Object of this class fills in the space provided by the JFrame. It also looks after the placement of all other user interface components.
* **JButton**. 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
```

Course material may not be reproduced in whole or in part
without the prior written permission of IBM.

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 editor open the file RPGProcFrame.java
> **x:\CODELAB\RPGProcFrame.java**

2b. The next few pages contain the 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 name
    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. Read through the code. Try to find all the pieces we talked about.

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

<div align="center">

SAVE x:\WDSC\JAVA\RPGProcFrame.java

</div>

and then pressing **Enter**.

Step 3. Using CODE to compile your Java classes

PURPOSE

The CODE editor provides a set of Verify/Compile/Debug actions for any supported iSeries language including Java. However, Java classes can run on your PC and on your iSeries. CODE targets both: one for Lpexlet development and the other for iSeries Java development. We therefore provide two sets of Compile/Run/Debug actions: local and remote.
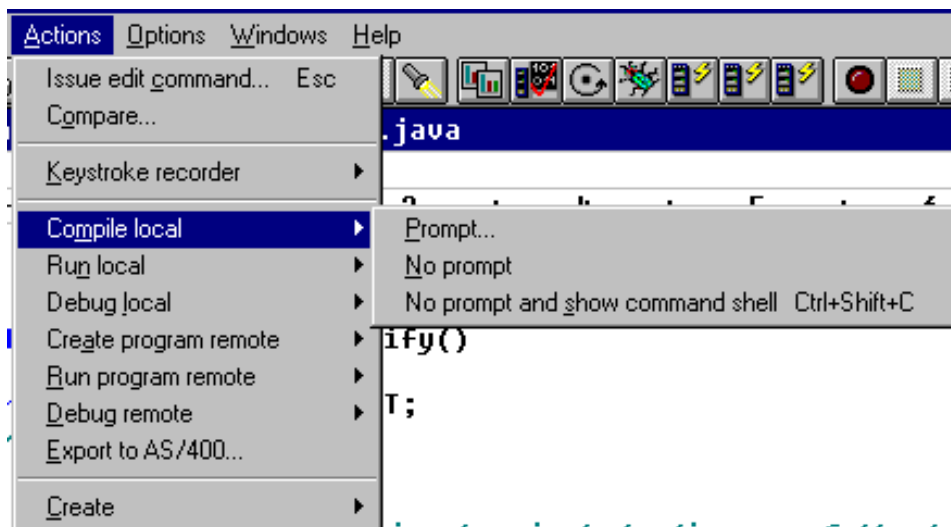
In this exercise we are developing Lpexlets and will therefore concentrate on local actions.

INSTRUCTIONS

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

3b. From the editor's '**Actions**' menu select the **'Compile local' -> 'Prompt...'** option.
After a few seconds (be patient - this is Java) the following dialog comes up.
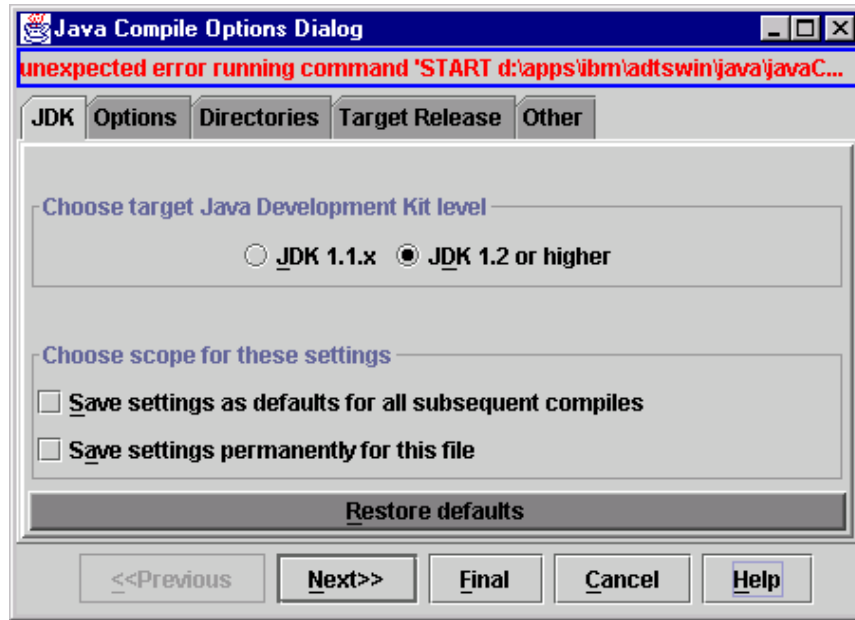
This dialog has several pages of Java compiler settings. You can use the 'Next>>' and 'Previous>>' buttons to navigate between pages. Get familiar with the dialog. You will need to use it quite a bit once you get into serious Lpexlet development!

3c. The defaults are just fine for now. Press the '**Final**' button and watch how RPGProcFrame class gets compiled. You will notice a 'Compiling...' 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.

3d. In the CODE editor switch to the **RPGProc.java** file.

3e. 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 one piece of the puzzle 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 press 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 in x:\wdsc\macros.

```
/**********************************************/
/* 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 ready now and we can start testing the Lpexlet.
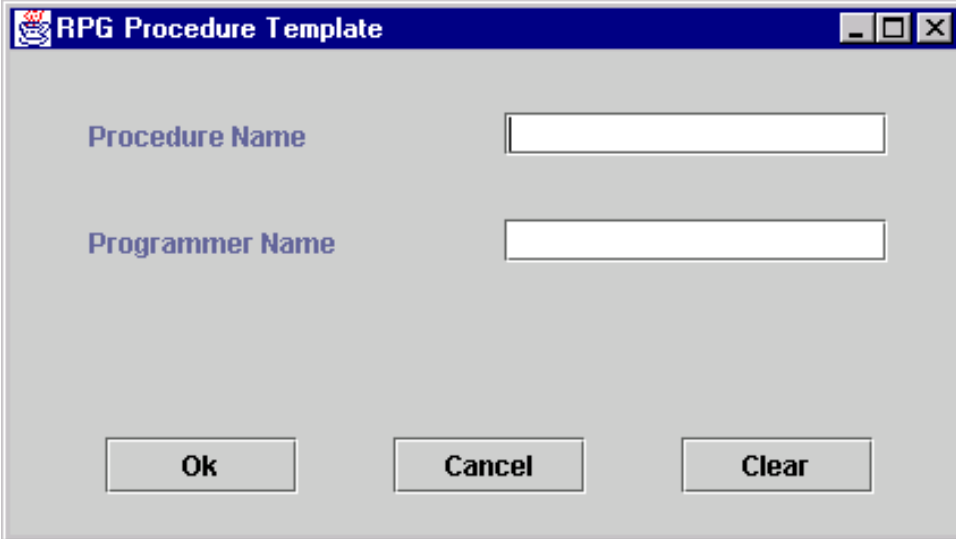
4d. Open a new ILE RPG file COMMON2.RPG by typing: **LX COMMON2.RPG** on the editor command line and then press the **Enter** key.

4e. Go to the editor command line and type:
<p style="text-align:center">**RUNJAVA RPGProc**</p>
and then press the **Enter** key.
Note the **case is important** when you call a 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
00001      D* ----------------------------------------
00002      D* Prototype for procedure: MyProc
00003      D* ----------------------------------------
00004      D MyProc          PR
00005
00006      P* ----------------------------------------
00007      P* Procedure Name: MyProc
00008      P* Purpose:
00009      P* Written by:    MyName
00010      P* ----------------------------------------
00011      P MyProc          B
00012      D MyProc          PI
00013
00014      C* Your calculation code goes here
00015
00016      C                    RETURN
00017      P MyProc          E
```

Notice that the 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 the user for other important pieces of information.

4g. From the **'File'** menu select '**Exit**' to close the CODE editor.

<u>Optional Exercise: WebSphere Studio Workbench</u>

Most of the functionality of the CODE Project Organizer has been replaced by WebSphere Studio functionality with the exception of accessing ADM parts.

The **Remote Systems Explorer** is the replacement for PDM (Program Development Manager) on the workstation. It currently doesn't have all the functions of PDM but will over time be a full replacement for PDM.

The RSE allows you to:

- Simplify your work by giving you quick access to lists of iSeries libraries, objects, members, IFS files, Unix files, and local files.
- Use the context-sensitive pop-up menus on these lists to perform actions such as start the CODE Editor, CODE Designer, or Distributed Debugger or other common iSeries actions.
- Use the **Work with User Actions** option to create and manage your own user-defined actions and have them appear in the pop up menus.
- Use the command support to increase your productivity by allowing you to enter and repeat iSeries or local commands without switching to an emulator session.

In the following exercise you will use the **Remote Systems Explorer (RSE)** perspective to work with the iSeries objects that you used in the previous exercises. You will also see how easy it is to perform actions and define your own actions. In short, you'll see how the **Remote Systems Explorer** can organize and integrate your work and make that work easier.

Selecting an iSeries object in the RSE perspective

In the RSE perspective, you now need to get to the iSeries objects you want to work with
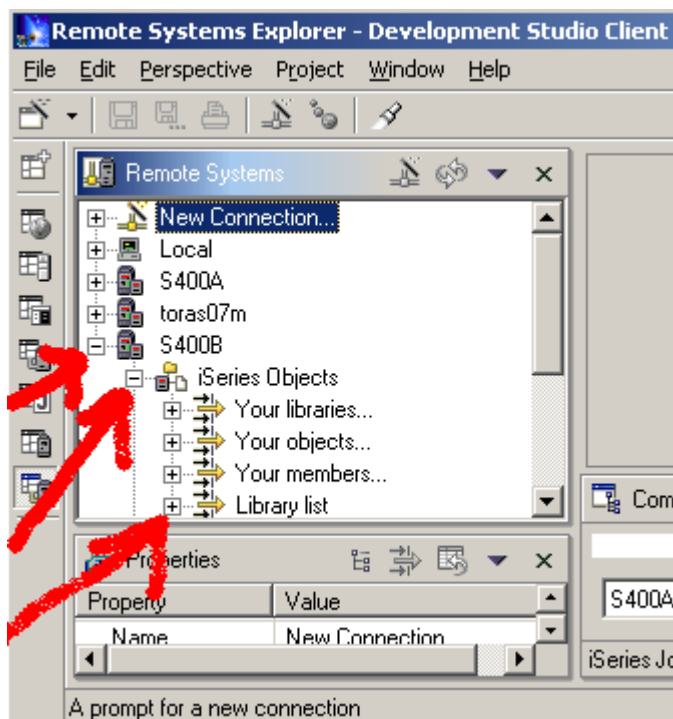
In the previous exercises you have worked with the library list filter, now you will create your own library filter.

First you will need to specify the library you want to work with:

- Expand the **connection node** that connects to your iSeries host, by clicking on the **+** sign beside it.

- Expand the **iSeries Objects** node.

To create a new library filter

- Expand the **Your libraries...** Node.

You see this dialog show up
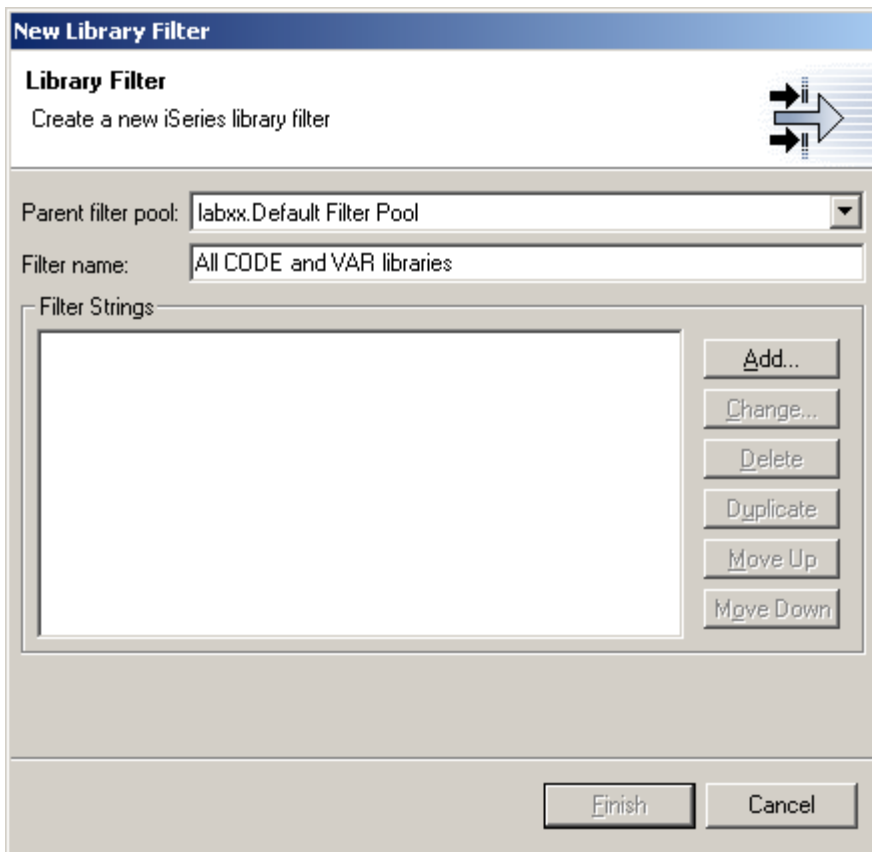
You are going to create a filter to specify the libraries you want to work with, so they will show in the RSE object list. We want you to create a filter that shows all libraries on the iSeries with the name **CODExxxxxx** and **VARxxxxxxx**, xxx being any character.

- Specify a name for this filter by keying into the **Filter name** entry field:

    All CODE and VAR libraries

- Press the **Add...** push button, beside the **Filter Strings** list

The following dialog will come up
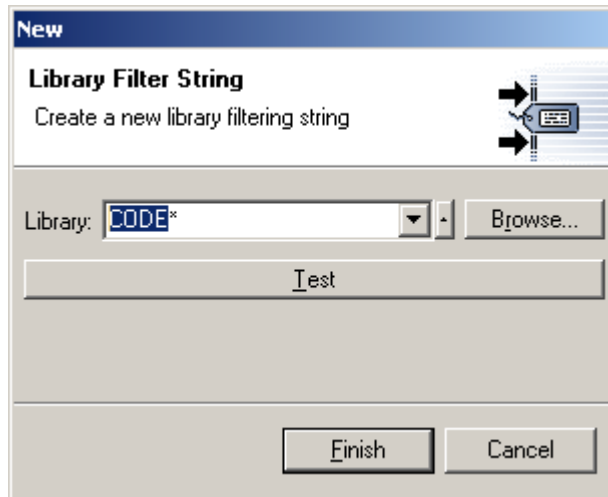
- Specify the first filter string that selects the libraries starting with CODE, by keying into the **Library:** entry field **CODE*** , using the * wild card character.

- Press the **Finish** push button

  Back in the **New Library Filter** dialog:

- Press the **Add** push button again.

  The **New Library Filter** string dialog shows up again.
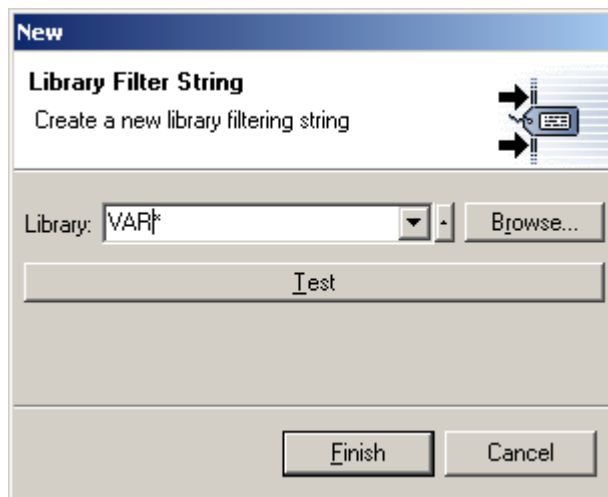
• This time, specify the Filter string to select all libraries starting with **VAR***

• Press the **Finish** push button.

The Library Filter dialog will now show the 2 Filter strings you specified:
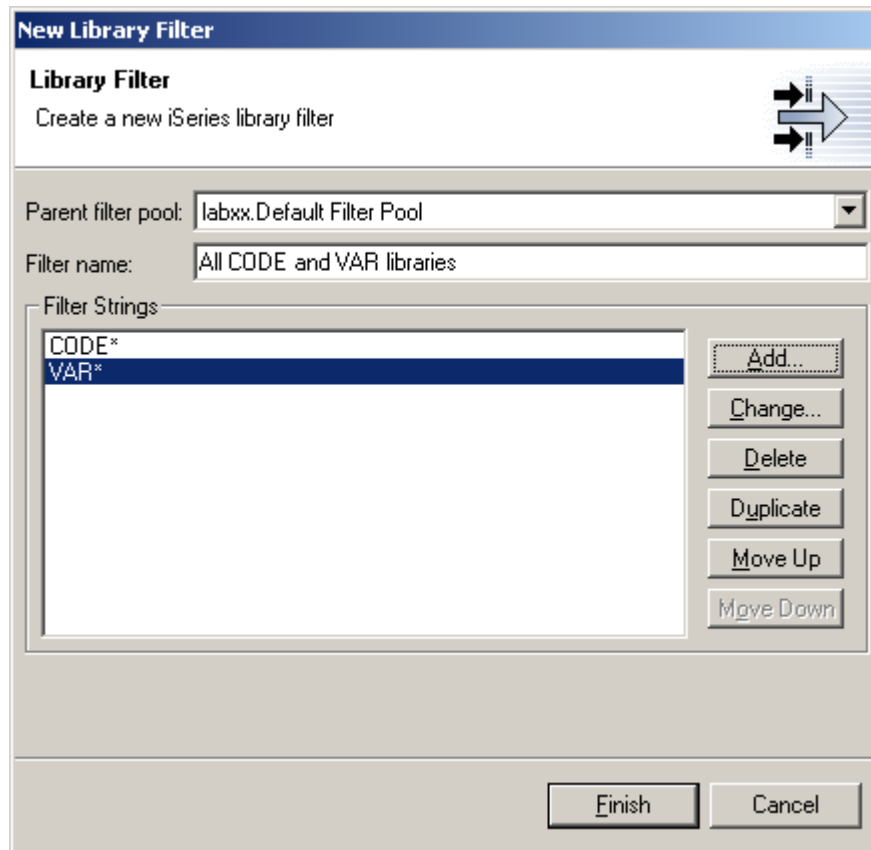
• Press the **Finish** push button on this dialog.

You are now back in the main workbench dialog.

You will see the list in the RSE perspective being expanded to include your filter.

• Expand **your new filter**.

If this is your first attempt to use this connection, you will be prompted to **Sign on** to the iSeries.
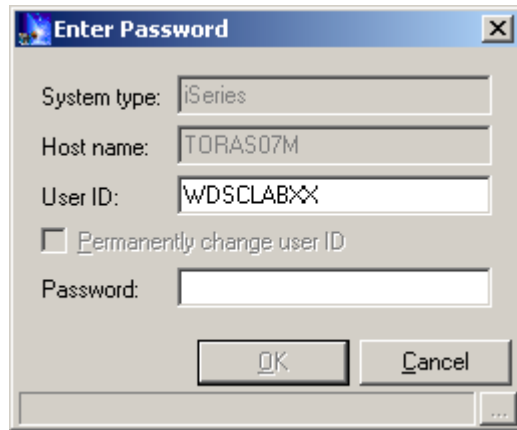
Use Userid **WDSCLABxx** and password **WDSCLABxx**

Back in the workbench in the RSE perspective you will see the libraries in your filter.
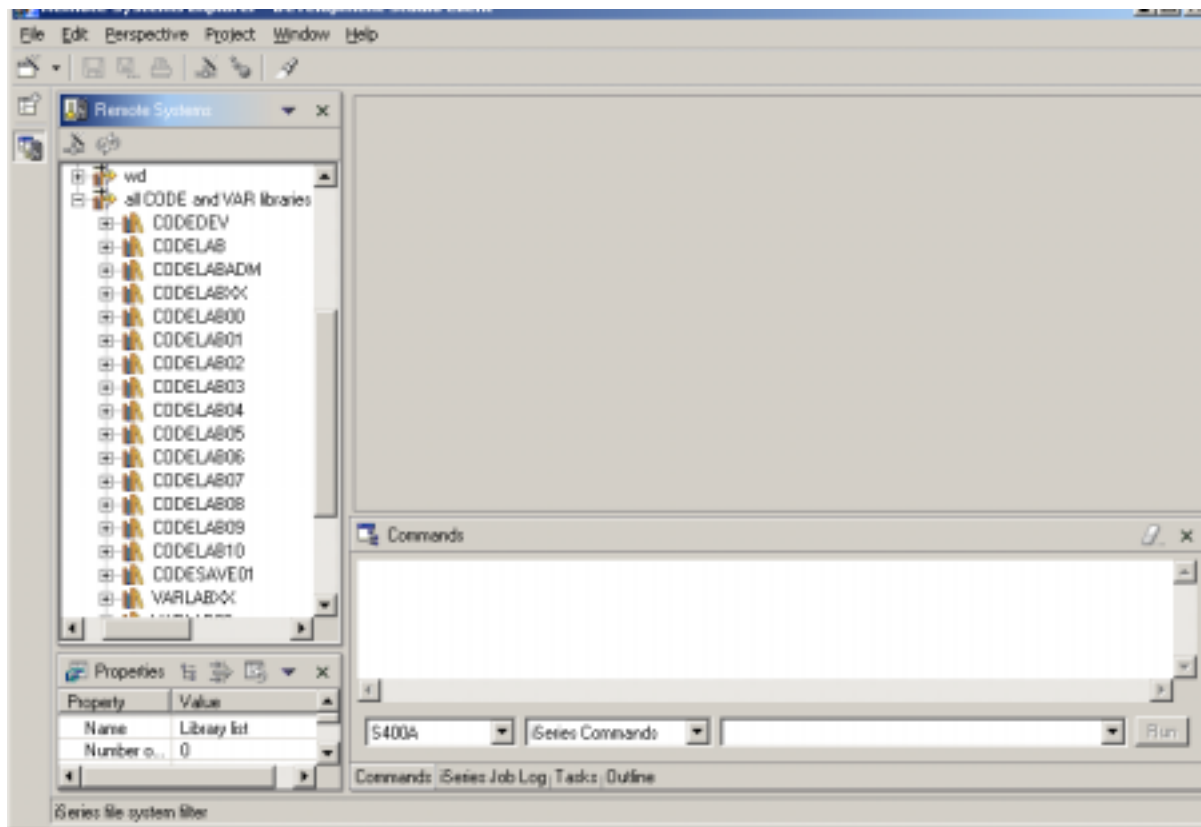
Now you can work with the libraries directly and can drill down to the object you want to work with.

Lets now create an object filter

In the RSE list, under the connection node you are using,
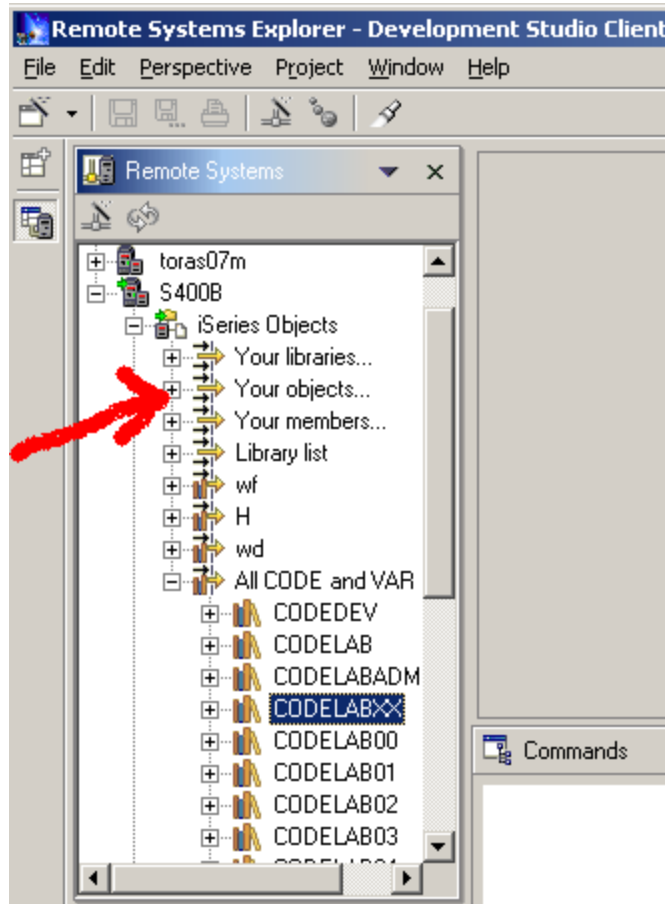
- Find the **Your objects...** Node,

- Expand it.

  This will show the New Object Filter dialog.

Create a filter to show all your source files in your **CODELABxx** library
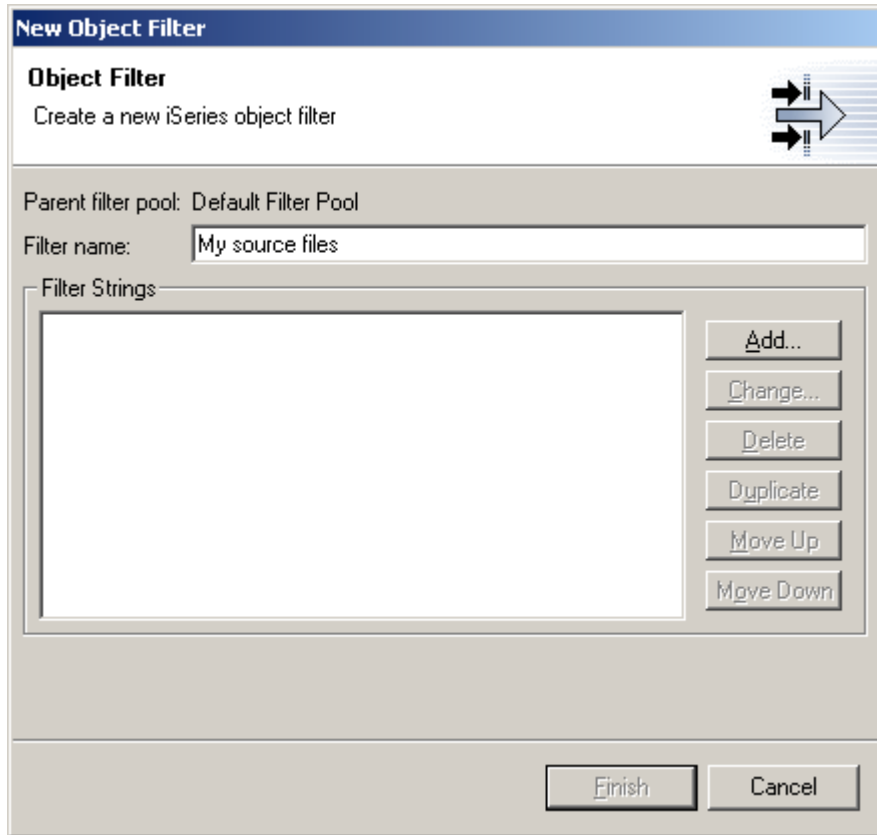
- Specify the **Filter name**: *My source files*

- Press the **Add...** Push button, beside the **Filter strings** list

In the Object Filter string dialog
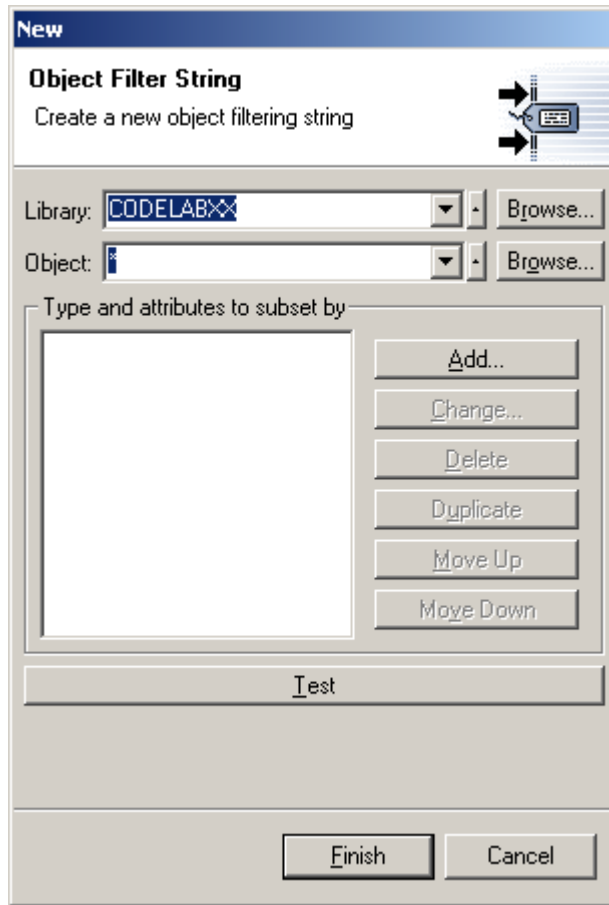
- Specify your **library**: *CODELABxx*

- For object leave the * wild card character since we want you to list all source files.

- Press the **Add...** Push button, beside the *Type and attributes to subset by* list .

In the **Add Object Type and Attribute** dialog
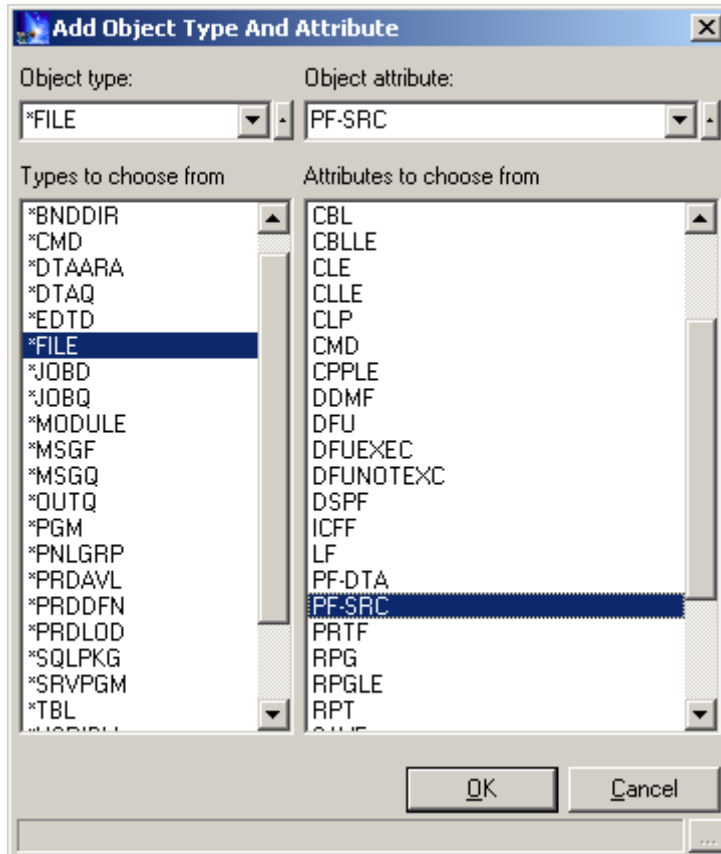
- Select ***FILE** as the Object type to be chosen.

- Select **PF-SRC** as the Object attribute to be chosen.

- Press the **OK** push button on this dialog.

- Press the **Finish** push button on the **Object Filter String** dialog.

- Press the **Finish** push button on the **New Object Filter** dialog.

The new filter will show up in the RSE list
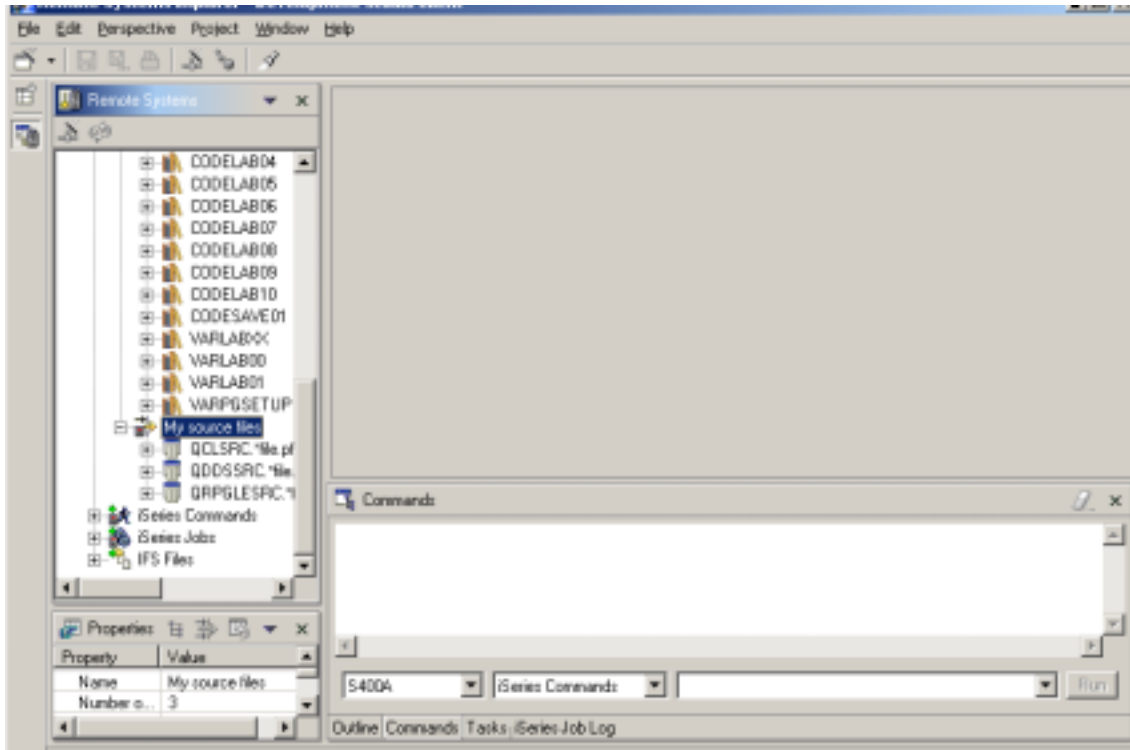
You got the idea how to create filters and tailor your development environment. Filters can also be specified for non iSeries servers and your local system.

Now you will work with the objects you have in your RSE list.

Assuming you want to edit the member **Payroll**, you just:

- Right mouse click on the member **Payroll**.

- From the pop up menu select **Open with**.

- From the sub menu select **LPEX editor**.

This will download the source member and open the editor with this member.

After you have edited the member you could save it and then compile it from the RSE list by using the pop up menu options on this member.

You can also create your own actions in addition to the default actions.

Creating a user action

In the RSE perspective, locate your **connection node** and the **iSeries Objects** node underneath it.
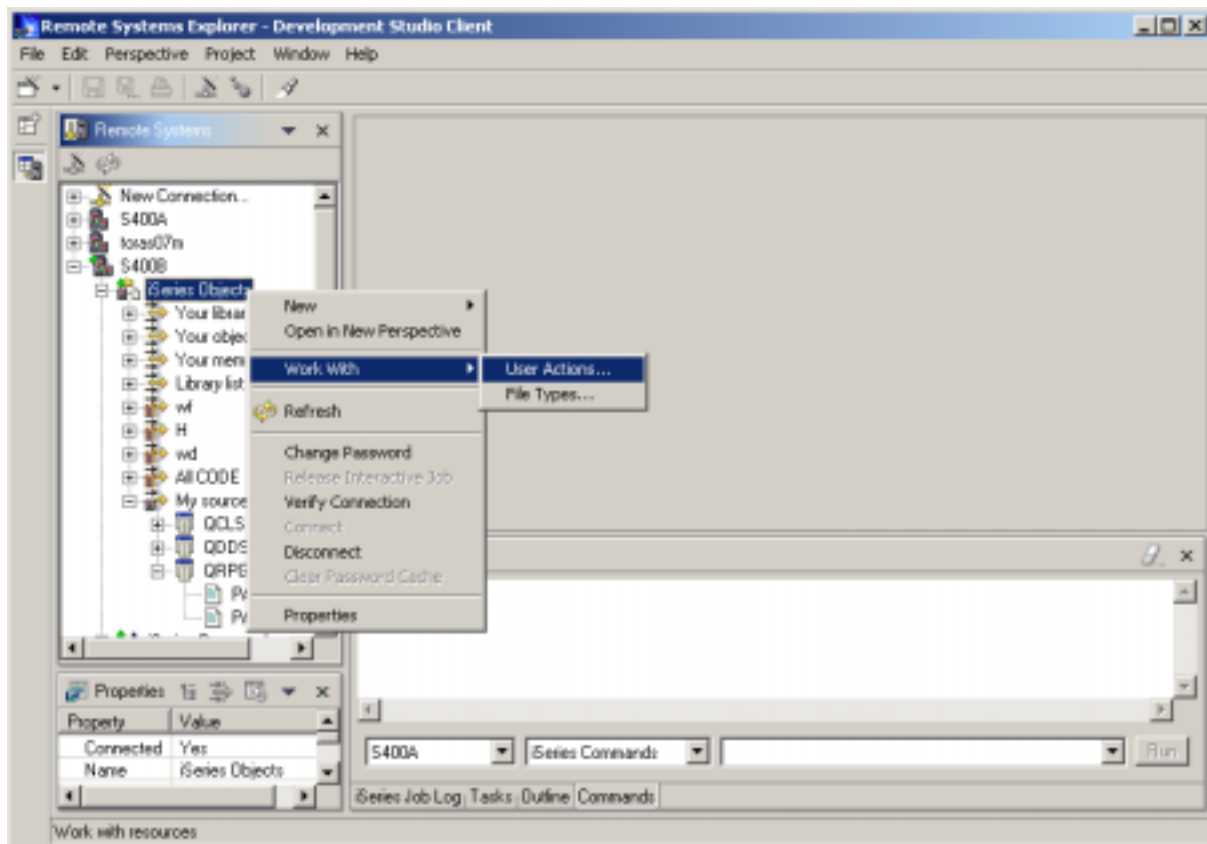
- Right mouse click on the **iSeries Objects** node.

- Select the **Work with** option from the pop up menu.

- Select the **User Actions** option from the sub menu.

You will see the **Work with User Actions** dialog
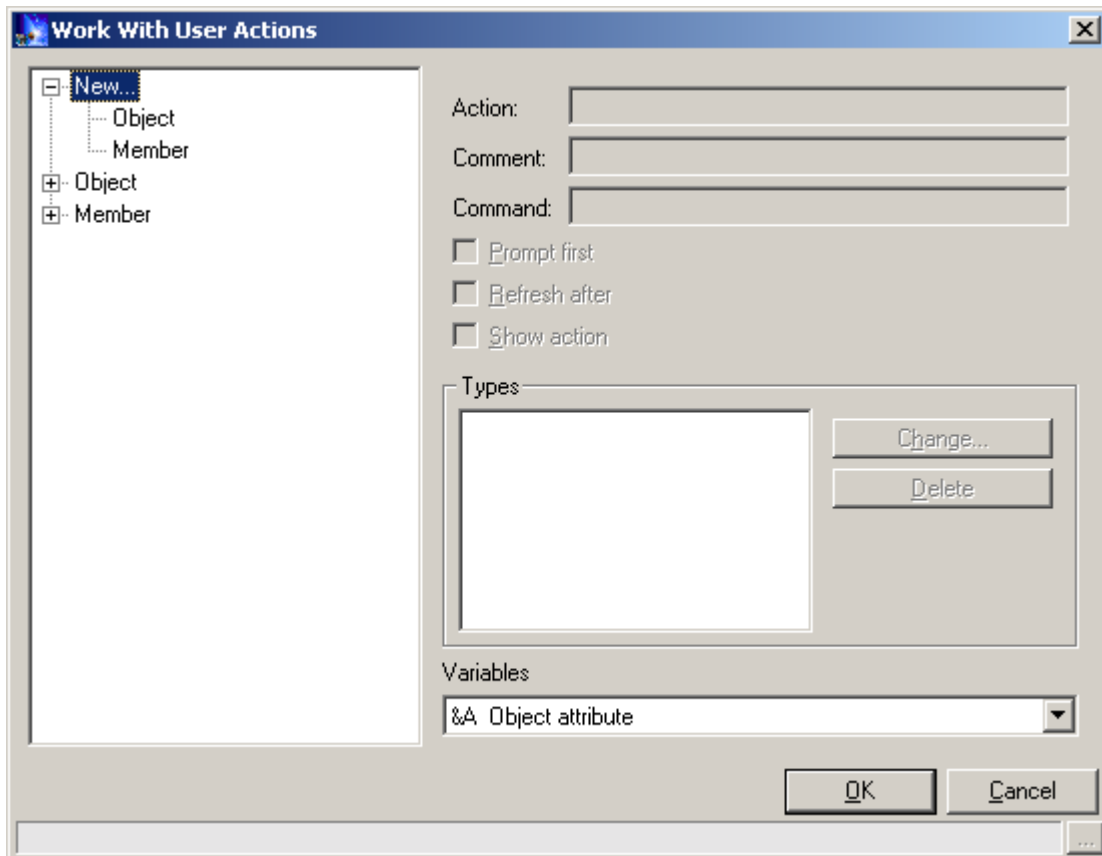
- Expand the **New** node in the list.

- Click on the **Object** node.

The New User Action dialog appears
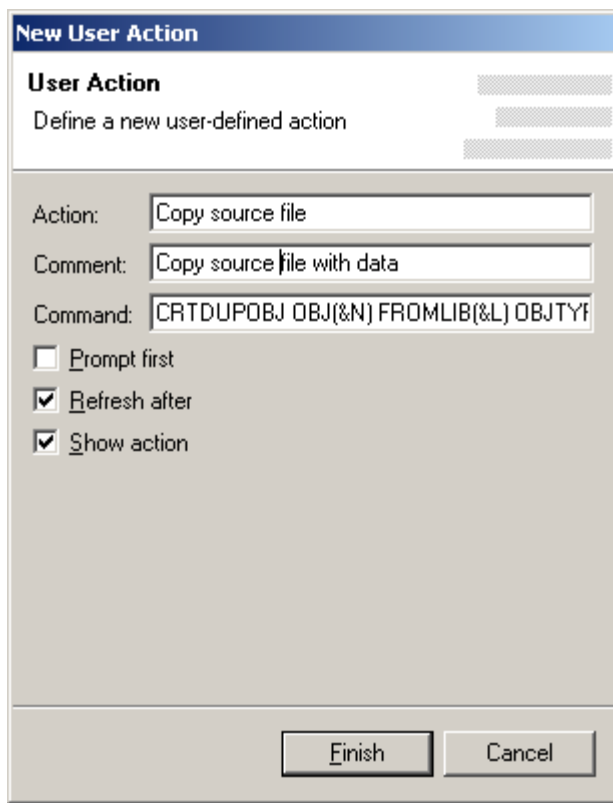
We want you to create a user action that copies a source file with data to a new source file.

- Enter a name for the user action in the **Action** entry field: ***Copy source file***

- Enter a comment in the **Comment** entry field.

- Key in the COMMAND to execute

  CRTDUPOBJ OBJ(&N) FROMLIB(&L) OBJTYPE(&T) NEWOBJ(QJUNKSRC) DATA(*YES)

  The name of the new source file is QJUNKSRC

- Select the **Refresh after** check box**.**

- Press the **Finish** push button.

You will be back at the **Work with User Action dialog**.

- Press the **Change...** push button, in the middle of the dialog.

The Type selection dialog will show up
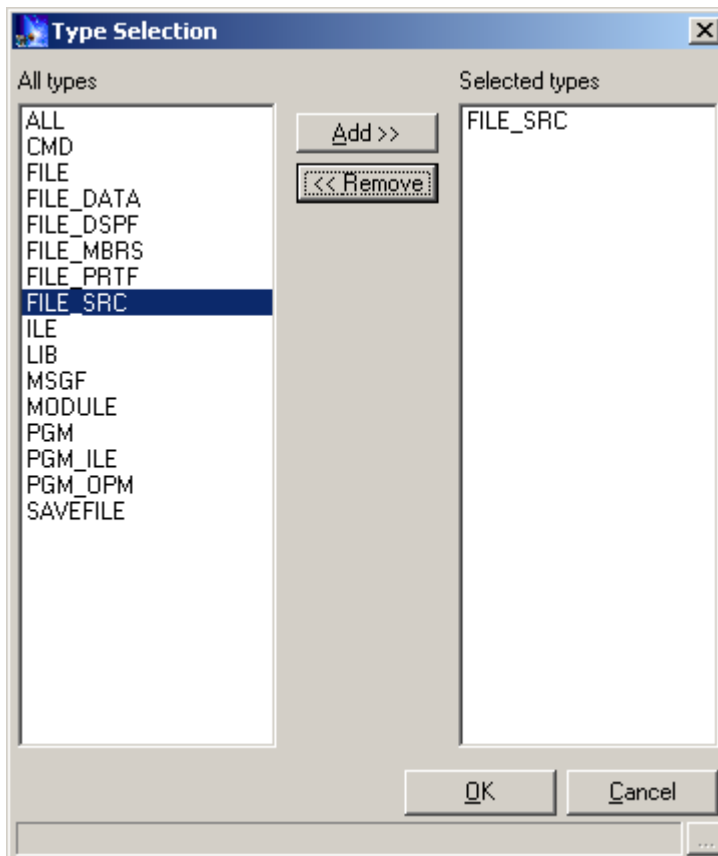
Since we want this **User Action** only to show up for **source files**, you will have to remove ***ALL** from the **Selected types** list and add **File_SRC** to it.

- Select **FILE-SRC** in the left list box.
- Press the **Add>>** push button in the middle between the list boxes to add it to the Selected types.
- Select ***All** in the right list box.
- Press the **<<Remove** push button in the middle between the list boxes.
- Press the **OK** push button.

Now, only when you right mouse click on a source file, will this user action appear on the pop up menu selected, for any other object type it will not appear.

Back in the workbench and the RSE perspective, give it a try.

- Locate your filter **My Source files.**
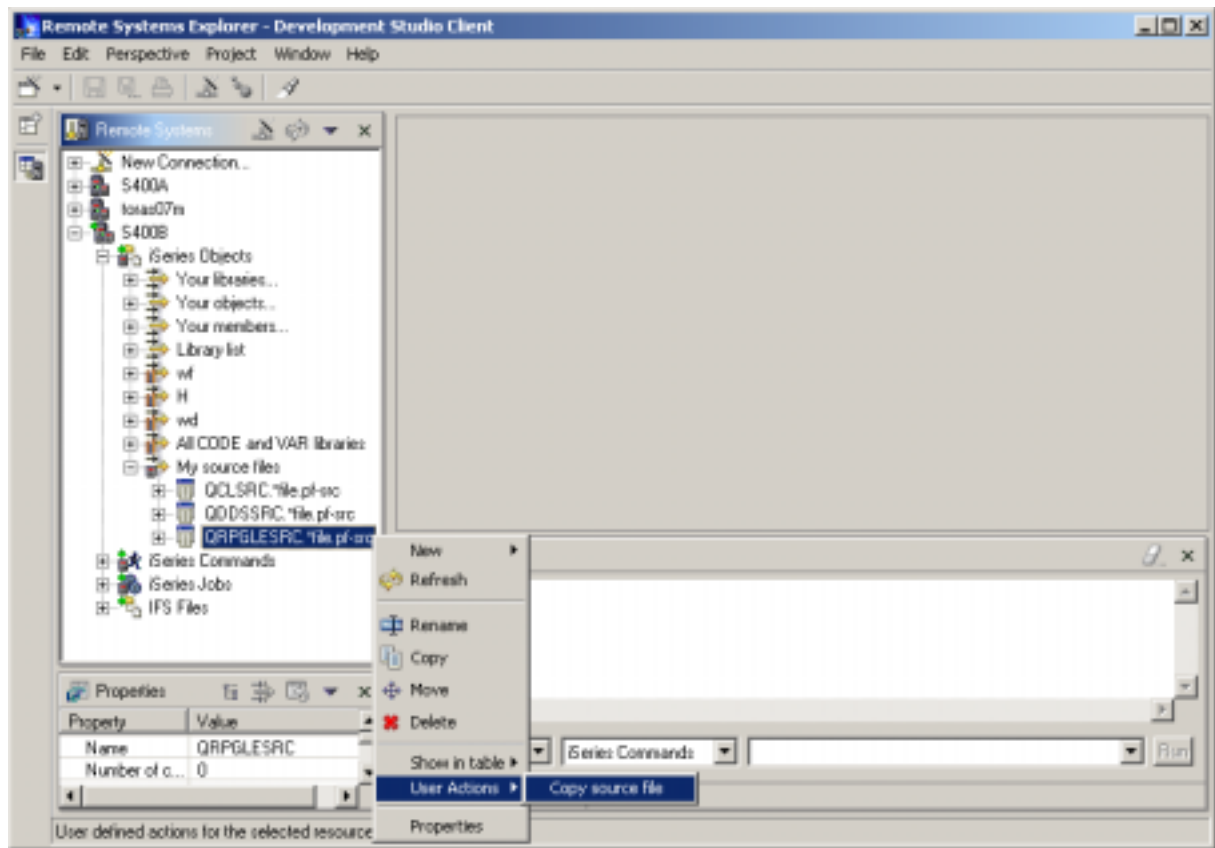
- Expand the **filter**, if it is not already expanded.

- Right mouse click on the **QRPGLESRC** file.

- Select **User Actions** from the pop up menu.

- Select **Copy source file** from the sub menu.

The file gets duplicated and the list gets refreshed, your new source file will show in the list.
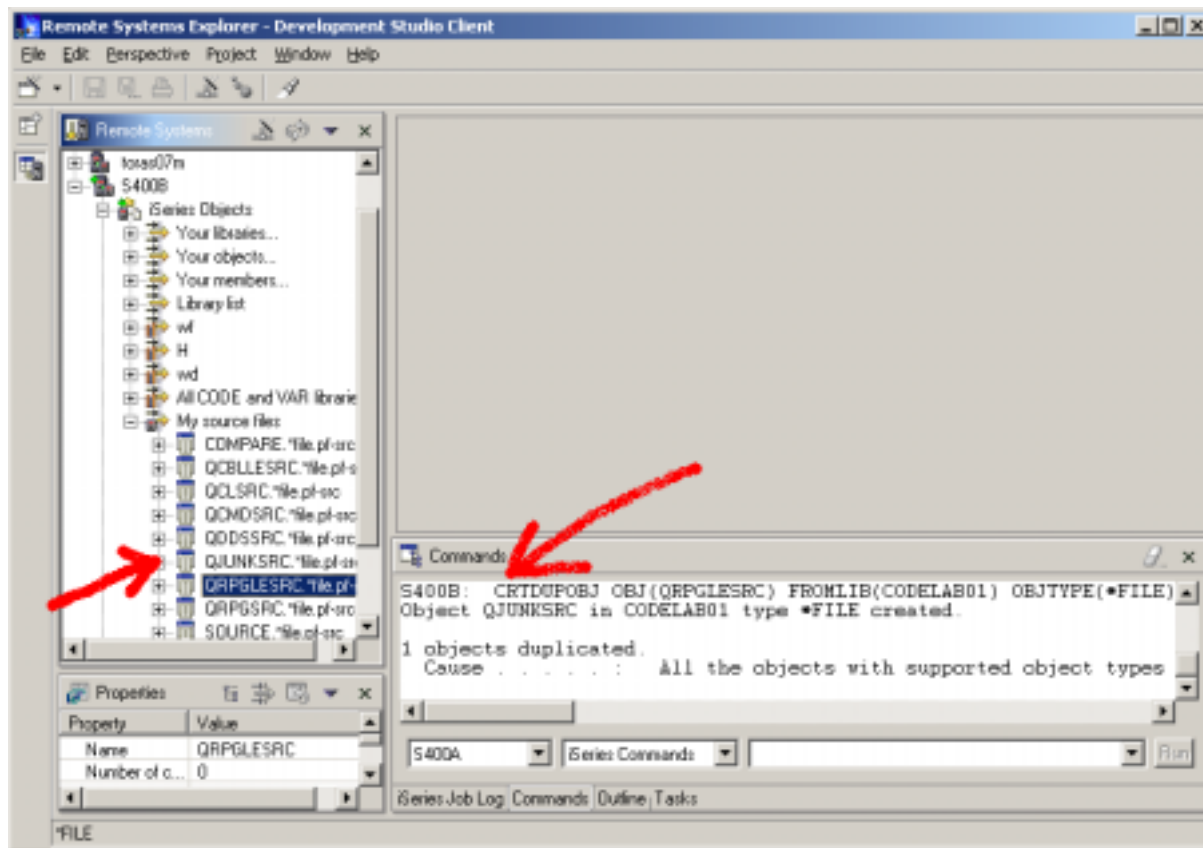
You can check the messages of the CL commands you are running in the RSE job by looking at the Commands view that by default shows up in the right hand side pane of the work bench. The arrow in the figure above points to it.

To delete the source file **QJUNKSRC** that you just created:

• Right mouse click on its **node** in the list.

• Select **Delete** from the pop up menu.

Running commands from the RSE

The Commands view should be up in the workbench
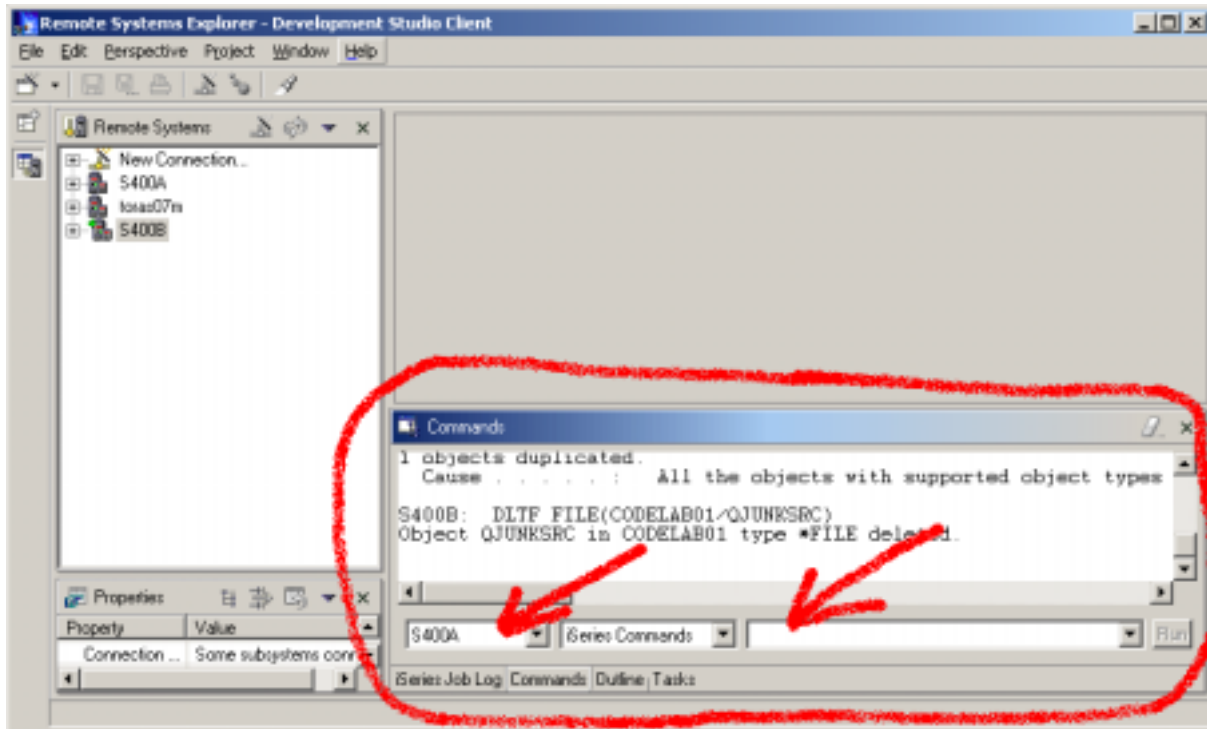
If not

- Click on the **Perspective** menu option in the workbench.

- Select **Show view** from the sub menu .

- Select **Commands** from the sub menu if it is available

    - Otherwise select **Other...** from the sub menu,

    - Expand the **Remote Systems** node in the **Show view** dialog,

    - Select **Command** in the expanded branch of Remote Systems,

    - Press the **OK** push button in the **Show view** dialog.

Now in the Command view you can select the iSeries server you want to run the command on, the left arrow in figure above points to the combo box.

- Select **your server** in the combo box.

- Key in an iSeries command for example **?ADDLIBLE**

The question mark is there to display a prompt screen
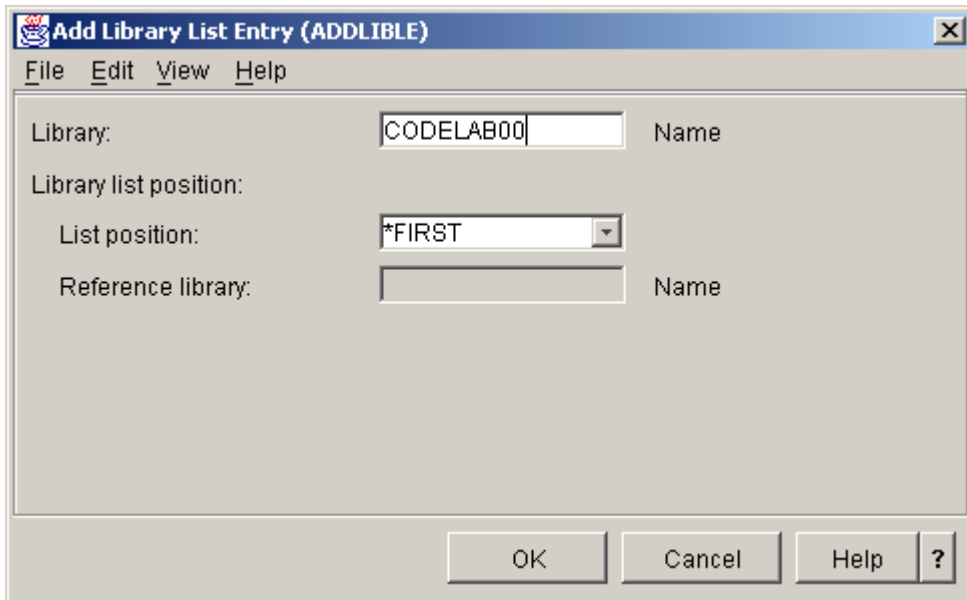
In the prompt dialog

- Key in **CODELAB00**, that will add this library to the library list of your RSE job on the iSeries server.

You could also use the **iSeries** commands node in the RSE view underneath the iSeries objects node and run predefined commands or define your own commands.

We hope this exercise gave you a first taste of the capabilities the RSE perspective provides to iSeries Application Developers.

*** Congratulations! ***

You have successfully completed the Advanced CODE lab. Programming the CODE editor may have left you bewildered, but you made it. Soon enough you will impress your boss and colleagues with some cool extensions to the CODE editor!

<u>Appendix A - The RPG Procedure SmartGuide</u>

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

One of the samples that comes with CODE 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:

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.

<u>Appendix B</u>

Extending the LPEX Editor in WebSphere Development Studio Client v4.0

In the CODE Editor of WebSphere Development Studio Client v4.0 (WDSC), users can customize their edit experience by adjusting the settings of the editor as well as by writing their own extensions to the editor. These extensions are commonly REXX macros or Java 'lpexlets'. The LPEX Editor in WDSC is based on the CODE Editor and also has the ability to be extended by users. Since WDSC is based on the Eclipse platform, these extensions are provided via plug-ins. Within a plug-in, Java classes which implement specific interfaces provided by the LPEX Editor are written to provide new functionality. The LPEX Editor itself is actually a plug-in as is other functionality provided in WDSC. This article will lead you through creating your own extension (a user command) to the LPEX Editor.

A brief outline of the steps that we will be progressing through:

> Creating your plug-in
> Creating your 'Hello World' user command
> Testing your 'Hello World' user command
> Adding another user command
> Packaging your plug-in

Note: Java classes are case-sensitive! It is important to enter text with the same case as specified in the steps.
Note: In some of the code examples, lines of the code wrap to the next line due to formatting of the document. They should not be on separate lines in the actual code. In most cases, you can just copy the code and paste it where needed.
Note: You do not need to know Java or Eclipse to follow the steps though it would be beneficial. You should be familiar with working with WDSC and the Remote Systems Explorer perspective in particular.

<u>Creating your plug-in:</u>

In Eclipse, functionality is provided via plug-ins. So, after you start WDSC, select **Perspective>Open>Other...** and select 'Plug-in Development Environment' to open a perspective which is designed for people who want to develop plug-ins. Then, select **File>New>Project**. In the New Project wizard, select 'Plug-in Development' in the left pane and then 'Plug-in Project' in the right pane.
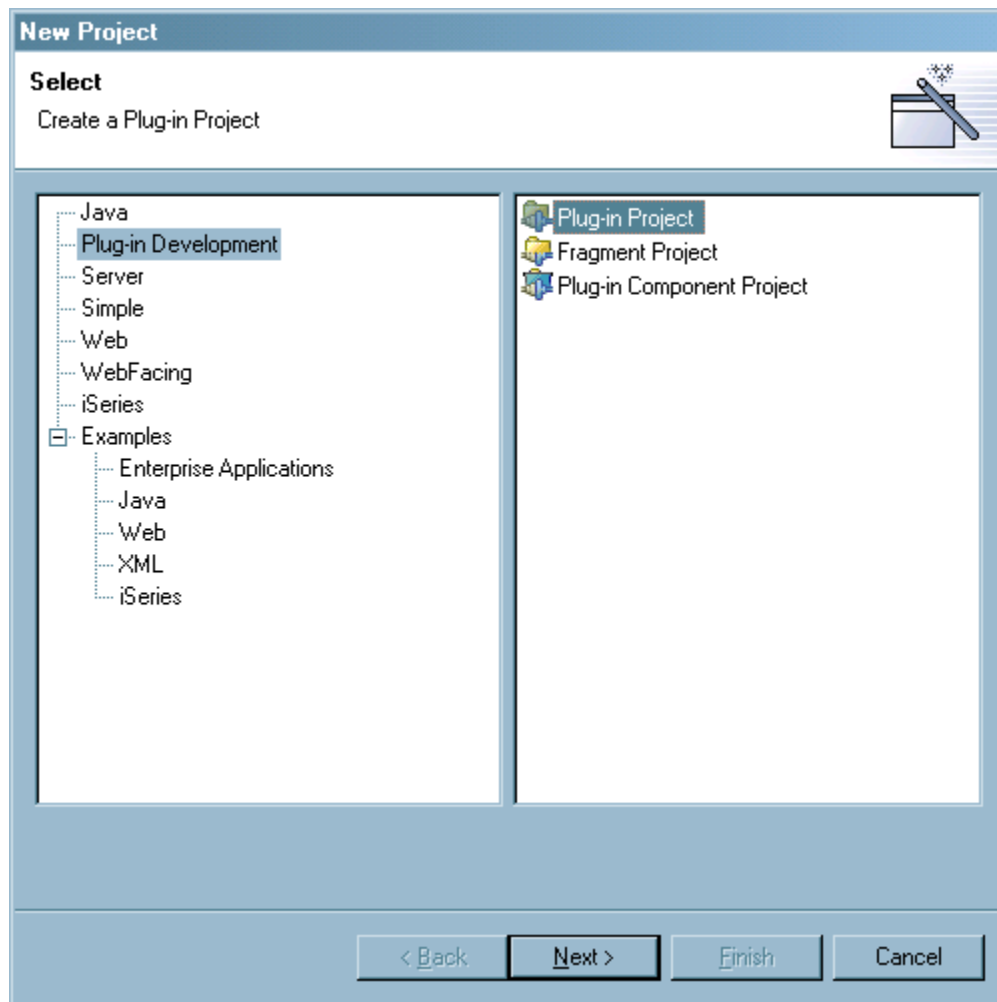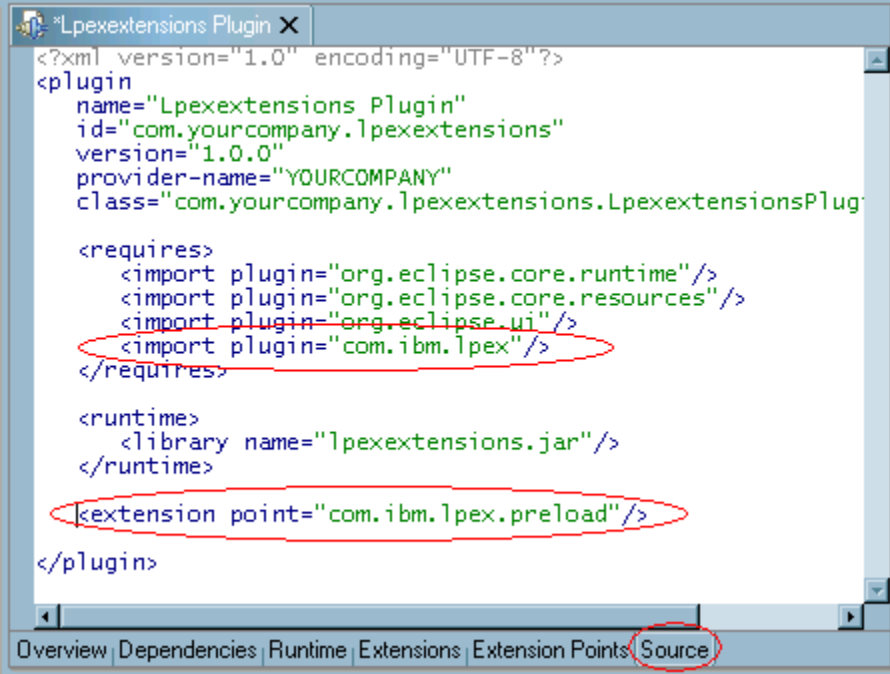
Click Next and on the 'Plug-in Project Name' page specify 'com.yourcompany.lpexextensions' as the project name. Click Next twice to get to the 'Plug-in Code Generators' page. Select the 'Create plug-in using a template wizard' and select the 'Default Plug-in Generator'. Then click Next. You can update the plug-in name and provider if you want. Then click Finish to create the plug-in and have it automatically generate the initial files required for a plug-in.

You should see your new plug-in (com.yourcompany.lpexextensions) in the Navigator view on the left side and your 'Lpexextensions Plugin' file open in the center.
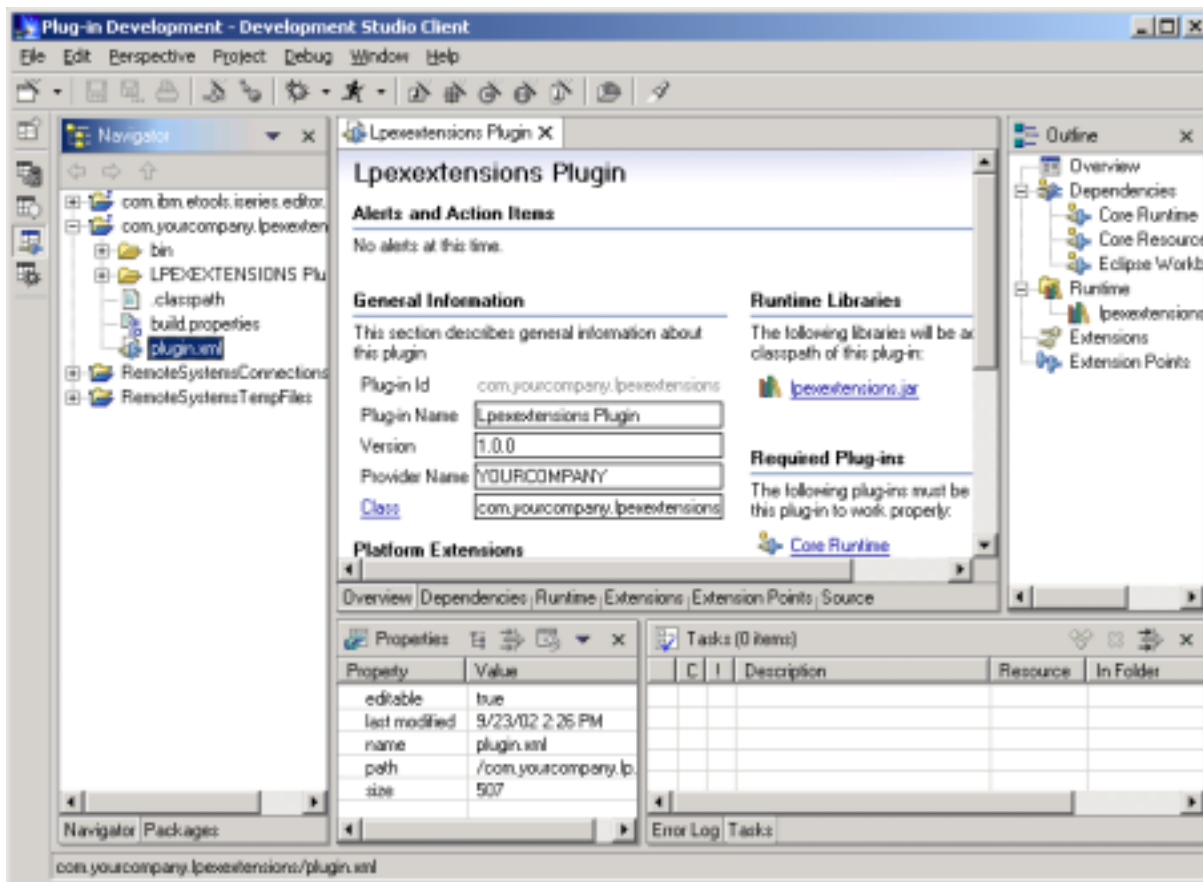
The 'Lpexextensions Plugin' that is open in the center is actually the 'plugin.xml' file that was generated for you by the New Project wizard. We need to update it so that it knows our plug-in requires the LPEX Editor plug-in and to register that we want to be able to contribute user commands to the LPEX Editor. To do this, go to the Source tab of the 'Lpexextensions Plugin' and add the lines indicated in the image.

Select **File>Save Lpexextensions Plugin** to save the file.

Now, we need to update the '.classpath' file which is in the plug-in. This file is used when the Java source which we will write is compiled. We need to update the file to indicate that we will be calling some of the LPEX Editor classes. In the Navigator view, double-click on the '.classpath' file to open and then add the following line before the '`</classpath>`' which is at the bottom.

```
    <classpathentry kind="var"
path="ECLIPSE_HOME/plugins/com.ibm.lpex/lpex.jar"/>
```

The file should look like the following:

```
Lpexextensions Plugin    *.classpath X
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
    <classpathentry kind="src" path="LPEXEXTENSIONS Plugin"/>
    <classpathentry kind="var"
        path="ECLIPSE_HOME/plugins/org.eclipse.core.runtime/runtime.jar" sourcepath
    <classpathentry kind="var"
        path="ECLIPSE_HOME/plugins/org.apache.xerces/xerces.jar" sourcepath="ECLIPS
    <classpathentry kind="var"
        path="ECLIPSE_HOME/plugins/org.eclipse.core.resources_1.0.2/resources.jar"
    <classpathentry kind="var"
        path="ECLIPSE_HOME/plugins/org.eclipse.ui_1.0.2/workbench.jar" sourcepath="
    <classpathentry kind="var"
        path="ECLIPSE_HOME/plugins/org.eclipse.help/help.jar" sourcepath="ECLIPSE_H
    <classpathentry kind="var"
        path="ECLIPSE_HOME/plugins/org.eclipse.swt_1.0.2/swt.jar" sourcepath="ECLIP
    <classpathentry kind="var"
        path="ECLIPSE_HOME/plugins/org.eclipse.core.boot/boot.jar" sourcepath="ECLI
    <classpathentry kind="var" path="JRE_LIB" rootpath="JRE_SRCROOT" sourcepath="JR
    <classpathentry kind="output" path="bin"/>
    <classpathentry kind="var" path="ECLIPSE_HOME/plugins/com.ibm.lpex/lpex.jar"/>
</classpath>
```

Select **File>Save .classpath** to save the file.

These are all the updates that we need to do to configure the plug-in. The plug-in is now created and the next step is to create an LPEX Editor user command.
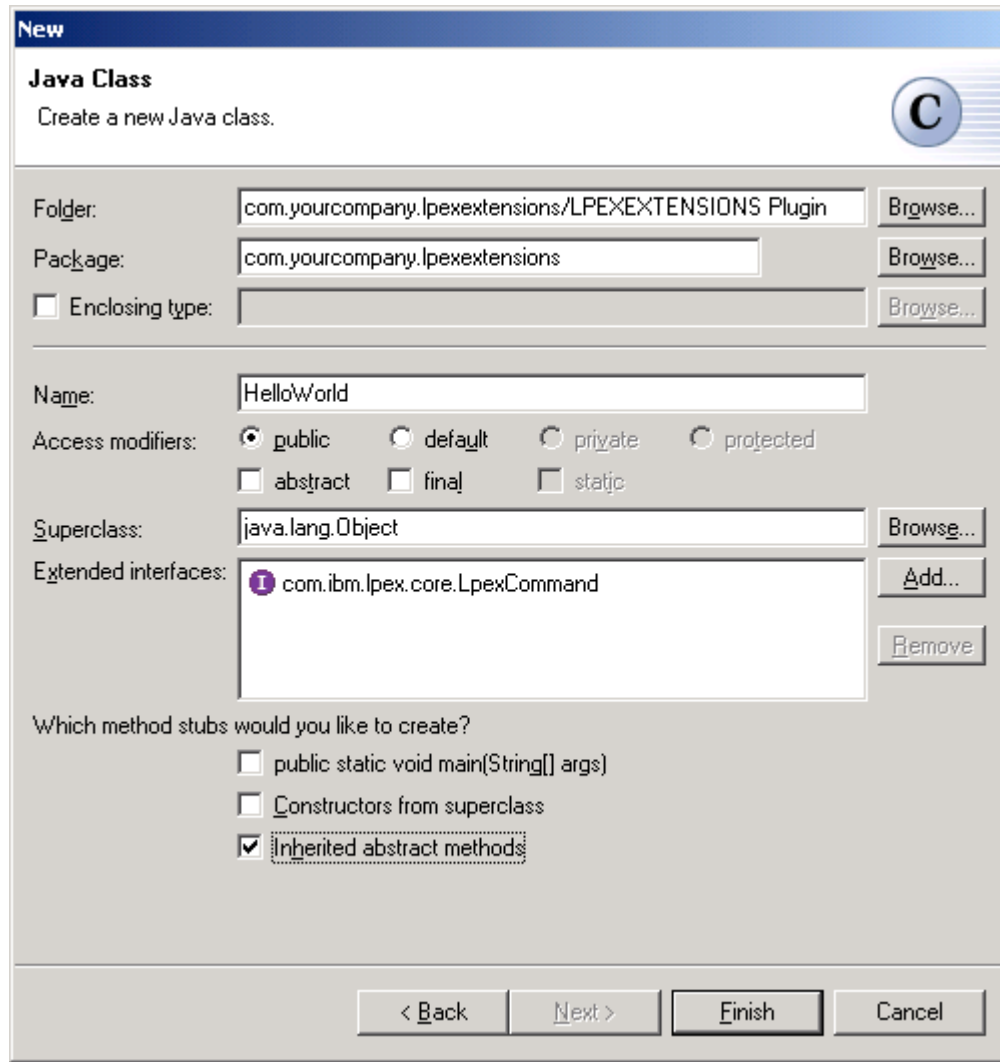
Creating the 'Hello World' user command:

For our first LPEX Editor user command, we will create one in the plug-in we just created which when run will display a 'Hello World!' message. Each LPEX Editor user command is a Java class which implements a specific interface. To create the class, select **File>New>Other...** In the New wizard, select 'Java' in the right pane and 'Java class' in the left pane.
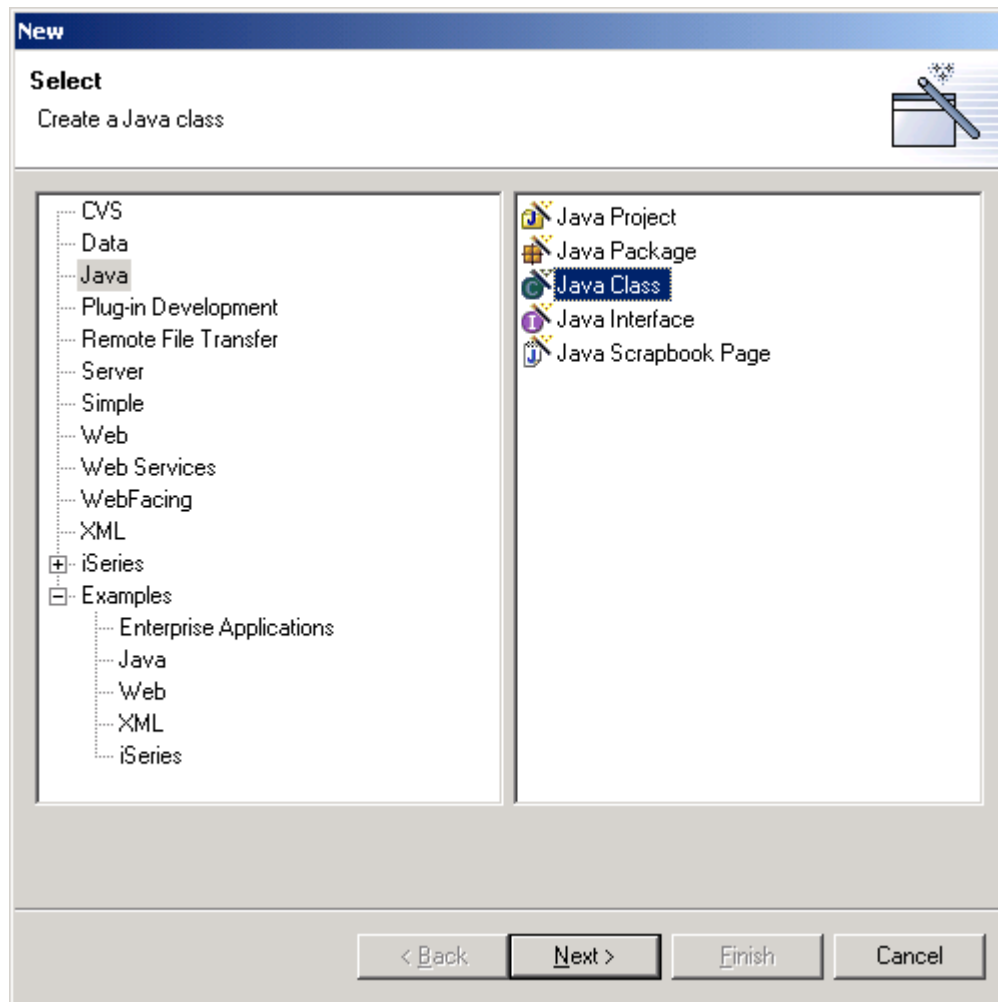
Click Next and specify the following information to create the new class:

     Folder: `/com.yourcompany.lpexextensions/LPEXEXTENSIONS Plugin`
     Package: `com.yourcompany.lpexextensions`
     Name: HelloWorld

Click on Add..., type 'LpexCommand' in entry field and then click OK. This should add 'com.ibm.lpex.core.LpexCommand' to the Extended Interfaces box. Java classes that implement the com.ibm.lpex.core.LpexCommand interface are LPEX Editor user commands. Then check the 'Inherited abstract methods' checkbox near the bottom of the page. The New Java Class wizard should look like the following:
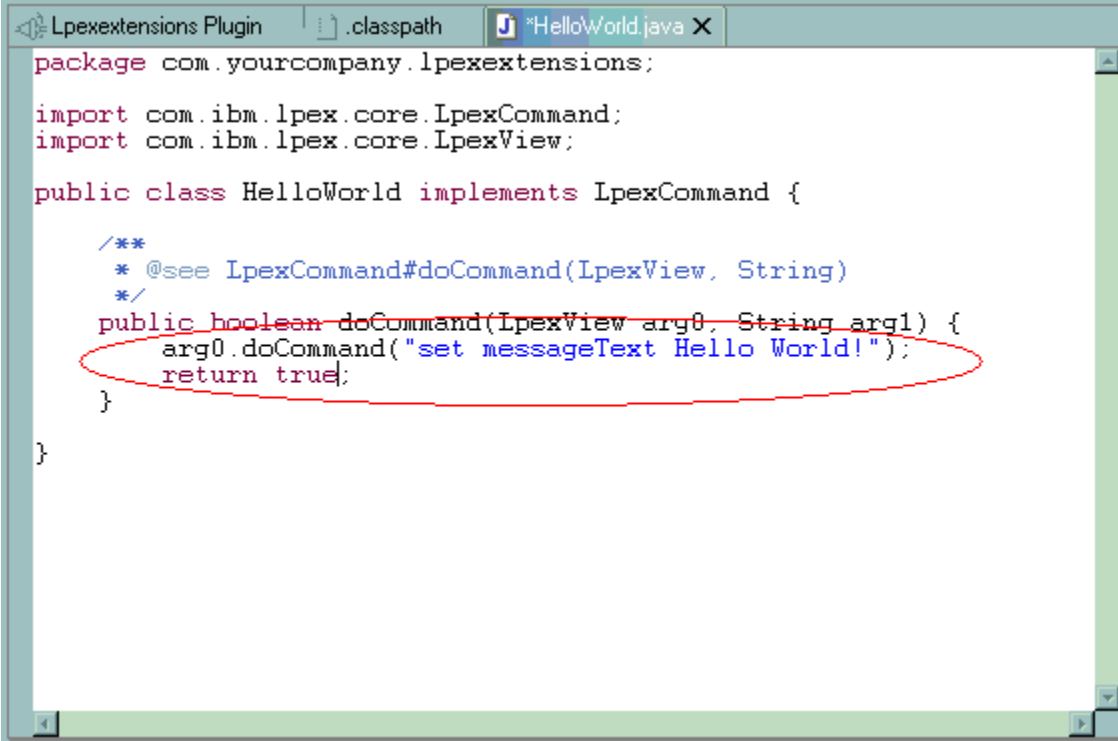
Click Finish to have the class created. The class should be automatically opened for you to edit it.
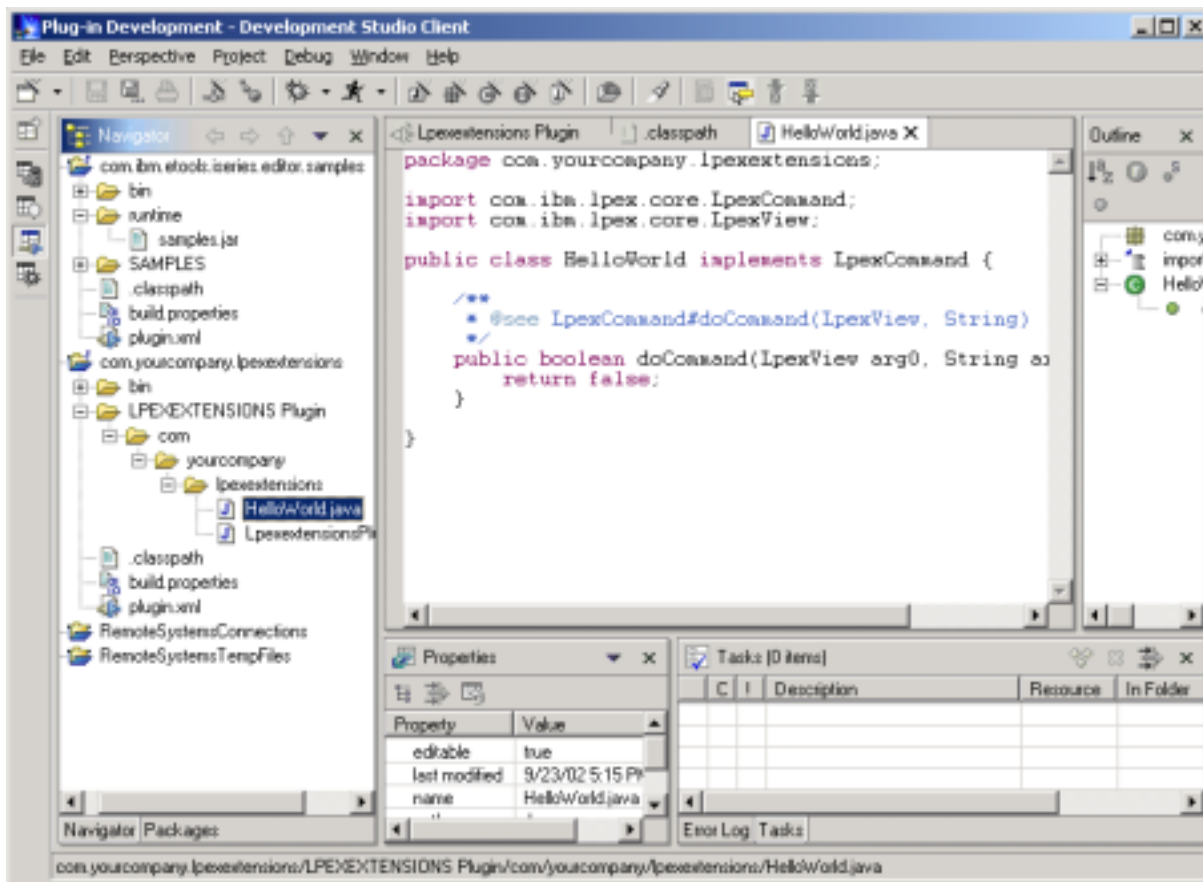
In an LPEX Editor user command class, the real work is done in the doCommand() method. This is where we will have the 'Hello World!' message issued. One of the input parameters to the doCommand() method is an LpexView object. With this object, we can call it's doCommand() which lets us run regular editor commands. In this case, we will use the 'set messageText' editor command to update the message line. So to issue the message, update the doCommand() method to the following:

```
Lpexextensions Plugin    .classpath    *HelloWorld.java X

package com.yourcompany.lpexextensions;

import com.ibm.lpex.core.LpexCommand;
import com.ibm.lpex.core.LpexView;

public class HelloWorld implements LpexCommand {

    /**
     * @see LpexCommand#doCommand(LpexView, String)
     */
    public boolean doCommand(LpexView arg0, String arg1) {
        arg0.doCommand("set messageText Hello World!");
        return true;
    }

}
```

Select **File>Save HelloWorld.java** to save the file. When the file is saved, it is automatically compiled. If there are errors, a ⊗ will appear beside the line with the error. If this happens, hover the mouse pointer over the ⊗ to have an error message shown. Now that the user commands exists, we still need to do one more step and that is to register it. In general, for each plug-in there is an actual corresponding Java class. It is this class that actually registers the user command. In this case, the plug-in class is the 'LpexextensionsPlugin.java' file which is in the same directory as our user command. If you do not see the file in the Navigator view, expand the 'LPEXEXTENSIONS Plugin' node under the 'com.yourcompany.lpexextensions' node until you see the file. Double-click on the file to open it.
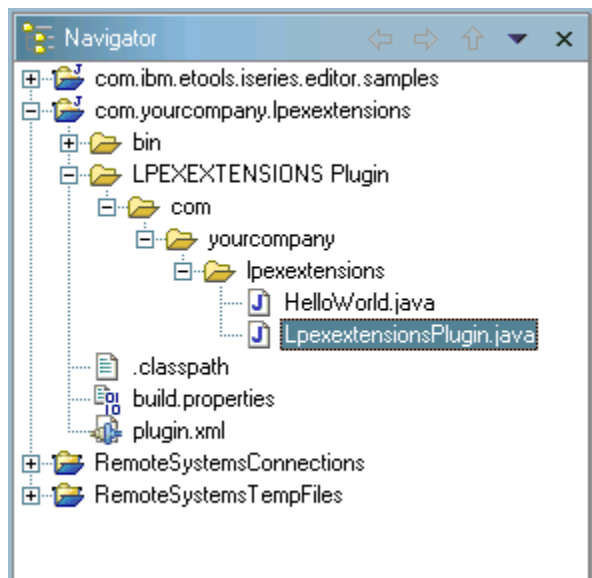
To register the user command that we just created we need to add a new method called 'startup'. The Eclipse platform will automatically call this method when the LPEX Editor is started because in our plug-in we registered that we want to add user commands. Add the following text before the last '}' in the file to add the method.

```java
        public void startup() throws CoreException {
                super.startup();
                String newUserCommands = "HelloWorld
com.yourcompany.lpexextensions.HelloWorld";
                String userCommands =
LpexView.globalQuery("current.updateProfile.userCommands");
                if ( userCommands == null )
                        LpexView.doGlobalCommand("set
default.updateProfile.userCommands " + newUserCommands);
                else if ( userCommands.indexOf(newUserCommands) < 0 )
                        LpexView.doGlobalCommand("set
default.updateProfile.userCommands " + newUserCommands + " " + userCommands);
        }
```

Also, we need to add '`import com.ibm.lpex.core.*;`' after the '`package com.yourcompany.lpexextensions;`' which is near the top of the file. After your changes, your class should be similar to the following:

```java
package com.yourcompany.lpexextensions;

import com.ibm.lpex.core.*;
import org.eclipse.ui.plugin.*;
import org.eclipse.core.runtime.*;
import org.eclipse.core.resources.*;
import java.util.*;

/**
 * The main plugin class to be used in the desktop.
 */
public class LpexextensionsPlugin extends AbstractUIPlugin {
        //The shared instance.
        private static LpexextensionsPlugin plugin;
```

```java
      //Resource bundle.
      private ResourceBundle resourceBundle;

      /**
       * The constructor.
       */
      public LpexextensionsPlugin(IPluginDescriptor descriptor) {
            super(descriptor);
            plugin = this;
            try {
                  resourceBundle=
ResourceBundle.getBundle("com.yourcompany.lpexextensions.LpexextensionsPluginR
esources");
            } catch (MissingResourceException x) {
                  resourceBundle = null;
            }
      }

      /**
       * Returns the shared instance.
       */
      public static LpexextensionsPlugin getDefault() {
            return plugin;
      }

      /**
       * Returns the workspace instance.
       */
      public static IWorkspace getWorkspace() {
            return ResourcesPlugin.getWorkspace();
      }

      /**
       * Returns the string from the plugin's resource bundle,
       * or 'key' if not found.
       */
      public static String getResourceString(String key) {
            ResourceBundle bundle=
LpexextensionsPlugin.getDefault().getResourceBundle();
            try {
                  return bundle.getString(key);
            } catch (MissingResourceException e) {
                  return key;
            }
      }

      /**
       * Returns the plugin's resource bundle,
       */
      public ResourceBundle getResourceBundle() {
            return resourceBundle;
      }

      public void startup() throws CoreException {
            super.startup();
            String newUserCommands = "HelloWorld
com.yourcompany.lpexextensions.HelloWorld";
            String userCommands =
LpexView.globalQuery("current.updateProfile.userCommands");
            if ( userCommands == null )
                  LpexView.doGlobalCommand("set
default.updateProfile.userCommands " + newUserCommands);
            else if ( userCommands.indexOf(newUserCommands) < 0 )
                  LpexView.doGlobalCommand("set
default.updateProfile.userCommands " + newUserCommands + " " + userCommands);
```

```
        }

}
```

Select **File>Save LpexextensionsPlugin.java** to save the file. When the file is saved, it is automatically compiled.

# Testing the 'Hello World' user command:

We have now written the user command and had it registered. Now you can test to see that it works. Eclipse provides a mechanism for you to test the plug-ins that you are developing. It actually enables you to start another WDSC window with your plug-in code activated. It does this by running your plug-in in the 'Runtime Workbench'. To do this, first select your plug-in in the Navigator view. Then select the down-arrow beside the running man toolbar button and then Run>Runtime Workbench.
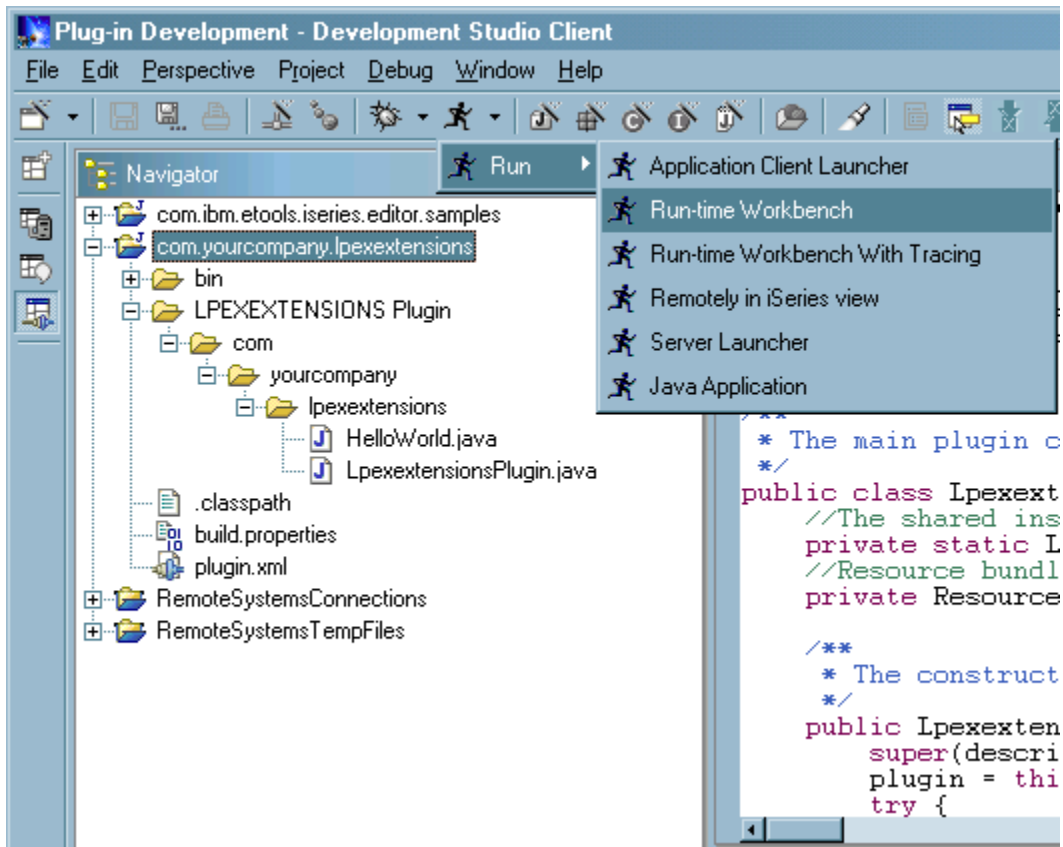

This will start a second WDSC window (in the first window, you will probably have the Debug perspective automatically opened, you can navigate back to your Plug-in Development

perspective using the [icon] icon along the left edge of the window). In the second WDSC window, open the Remote Systems Explorer perspective from the Perspective>Open>Other... dialog if it is not already open. Expand the 'Local Files' under the 'Local' connection. Under 'Local Files', expand 'Drives' and then expand the drive where WDSC is installed (usually C:).
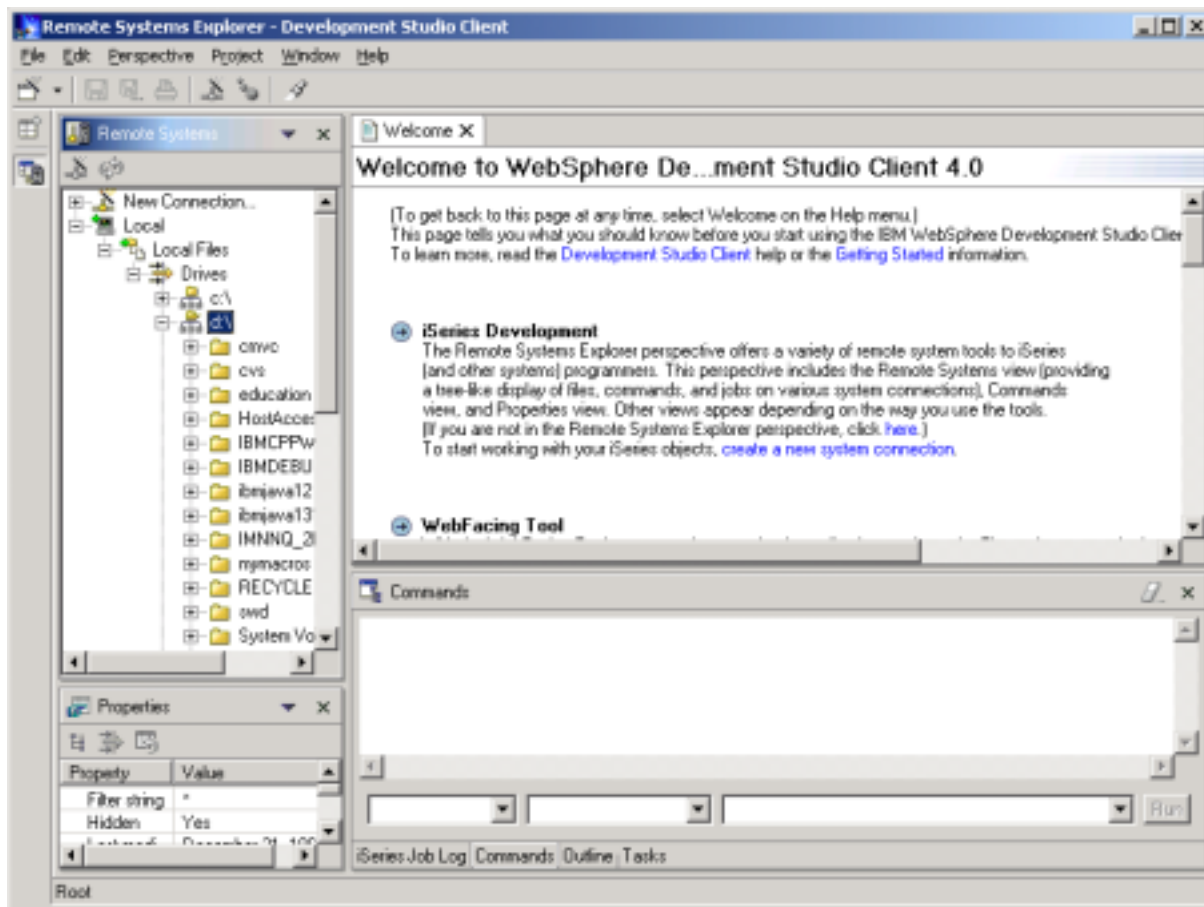
Then expand the WDSC directory. Scroll down to the 'sample.irp' file. Right-click on the file and select Open with>LPEX Editor. The file will be opened in the LPEX Editor. Now we just need to run our user command from the LPEX Editor command line to test it. The LPEX Editor command line is at the bottom of the editor view. Enter 'HelloWorld' in the command line.
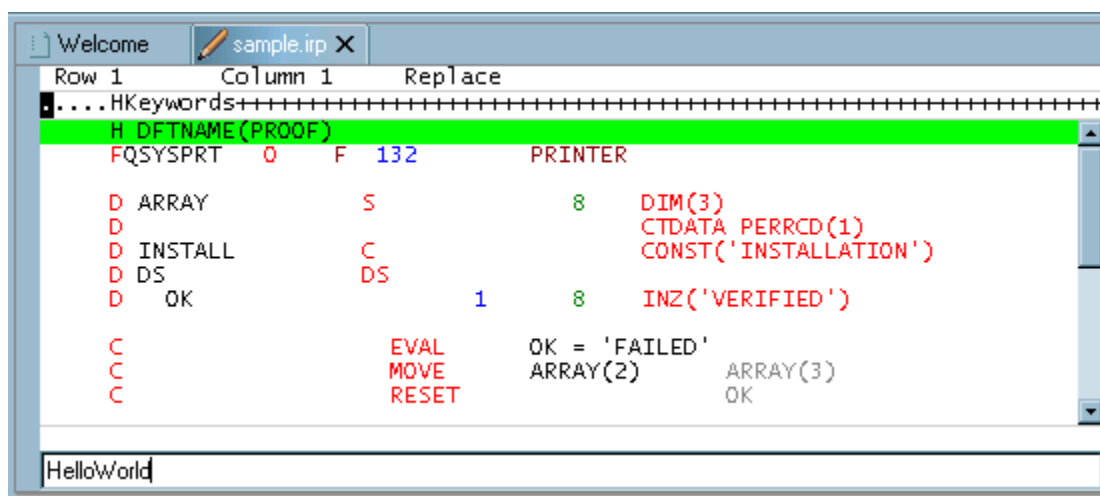
You should see the message 'Hello World!' displayed on the LPEX Editor message line!
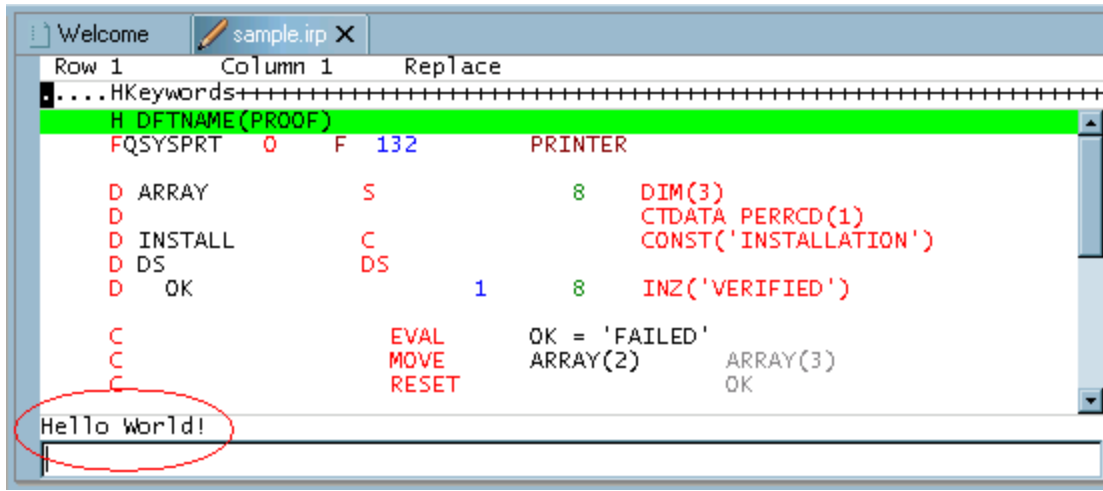
Adding another user command:

So far, you have created a plug-in, created a user command in the plug-in, and registered the user command. A plug-in can contain as many user commands as you want. You just need to create a class following the steps listed in the "Creating the 'Hello World' user command" section except give it a different name. Then you update the doCommand() method in the class for what you want to do. Once that is done, you update the startup() method in the LpexextensionsPlugin.java to register the user command.

Create a new user command and give it a name of 'Center'. In it's doCommand() method, use the following:

```java
/**
 * Center the text on the line
 *
 * @see LpexCommand#doCommand(LpexView, String)
 */
public boolean doCommand(LpexView view, String arg1) {

        // get the text of the current element
        int currElement = view.currentElement();
        String text = view.elementText(currElement);

        LpexLog.log("center: existing text is " + text );

        int length = 0;
        String textLimit = view.query("save.textLimit");
        if ( !textLimit.equals("default") )
                length = view.queryInt("save.textLimit");
        else
                length = text.length();

        LpexLog.log("center: maximum length of line is " + length );

        text = text.trim();
```

```
        int textlength = text.length();

        // center the text
        if (textlength < length) {
            int spaces = (length - textlength) / 2;
            if (spaces > 0) {
                int i = 0;
                String newtext = text;
                String blank = " ";
                for (i = 0; i < spaces; i++)
                    newtext = blank.concat(newtext);
                view.setElementText(currElement, newtext);
                LpexLog.log("center: new line is " + newtext);
            }
            view.doDefaultCommand("set messageText Line has been
centered.");
        } // end if text length < text limit

        // true indicates parameters are valid
        return true;

    }  // end of doCommand
```

Remember to save the file.

Update the startup() method LpexextensionsPlugin.java file to register your new user command. To do this, you just need to update one line of the method to also list the new command ('Center') and class ('com.yourcompany.lpexextensions.Center).

```
        String newUserCommands = "HelloWorld
com.yourcompany.lpexextensions.HelloWorld Center
com.yourcompany.lpexextensions.Center";
```

Remember to save the file. To test it, you should close the second WDSC window and then follow the steps in the "Testing the 'Hello World' user command" section except type 'Center' instead of 'HelloWorld' to run the new user command. The user command just centers text on the line but does demonstrate some of the capabilities that you have available in a user command. It queries information from the editor, performs some manipulation, logs messages to a file, and updates the editor's message line.

Packaging your plugin:

Now that you have a working plugin, you will want to be able to share it with others. So far, we have been running it in the Plug-in Development Environment which is used for testing your work. To share it with others we need to package it. The first step is the create a JAR file (which is essentially a zip file) for your Java classes that you have written. When we created the plugin, one of the settings was for the runtime library name which defaulted to 'lpexextensions.jar'. We need to create jar with this name in your plugin directory and have it contain your Java classes. To do this, select **File>Export**.
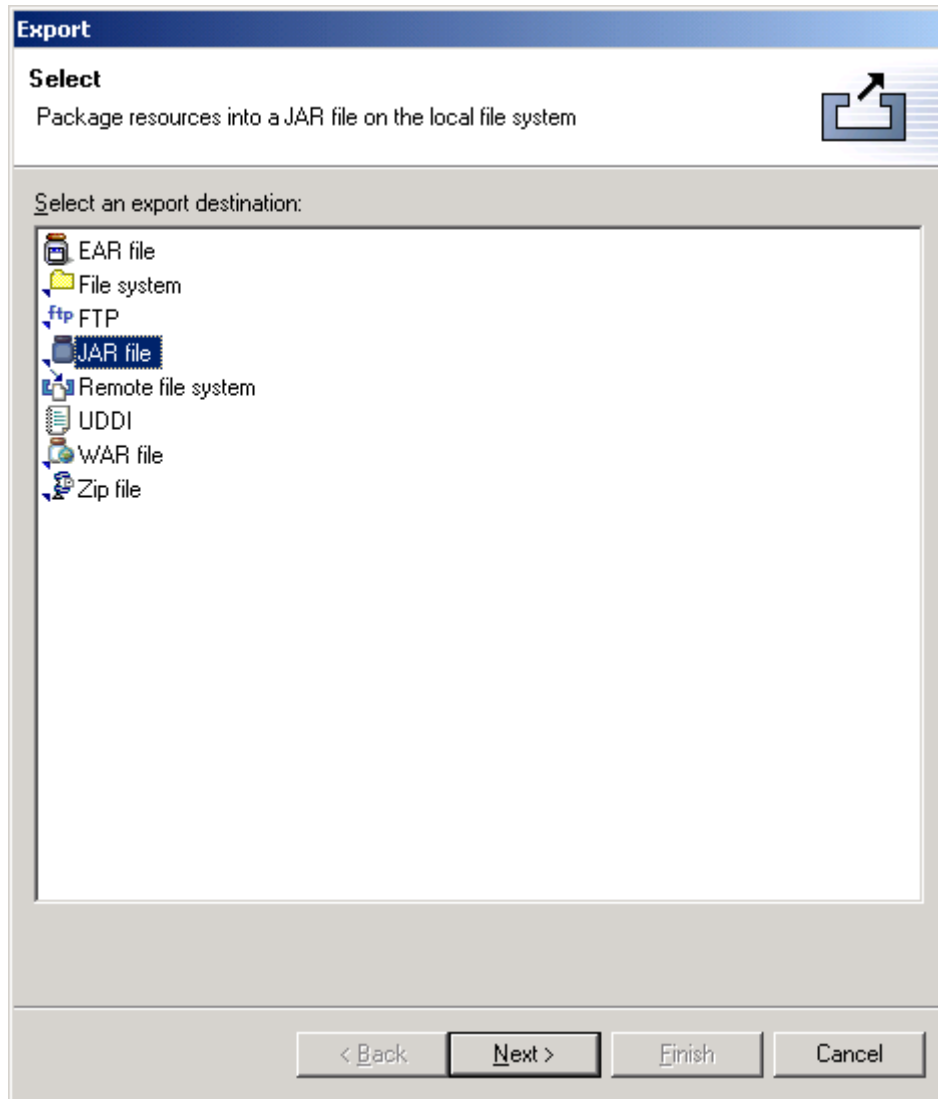
Select JAR file and click Next. On the JAR Package Specification page, expand the 'com.yourcompand.lpexextensions' node and click twice on the checkbox beside 'LPEXEXTENSIONS Plugin'. Then in the JAR file field specify 'c:\WDSC\WSSD\workspace\com.yourcompany.lpexextensions\lpexextensions.jar' if c:\WDSC is where you have WDSC installed.
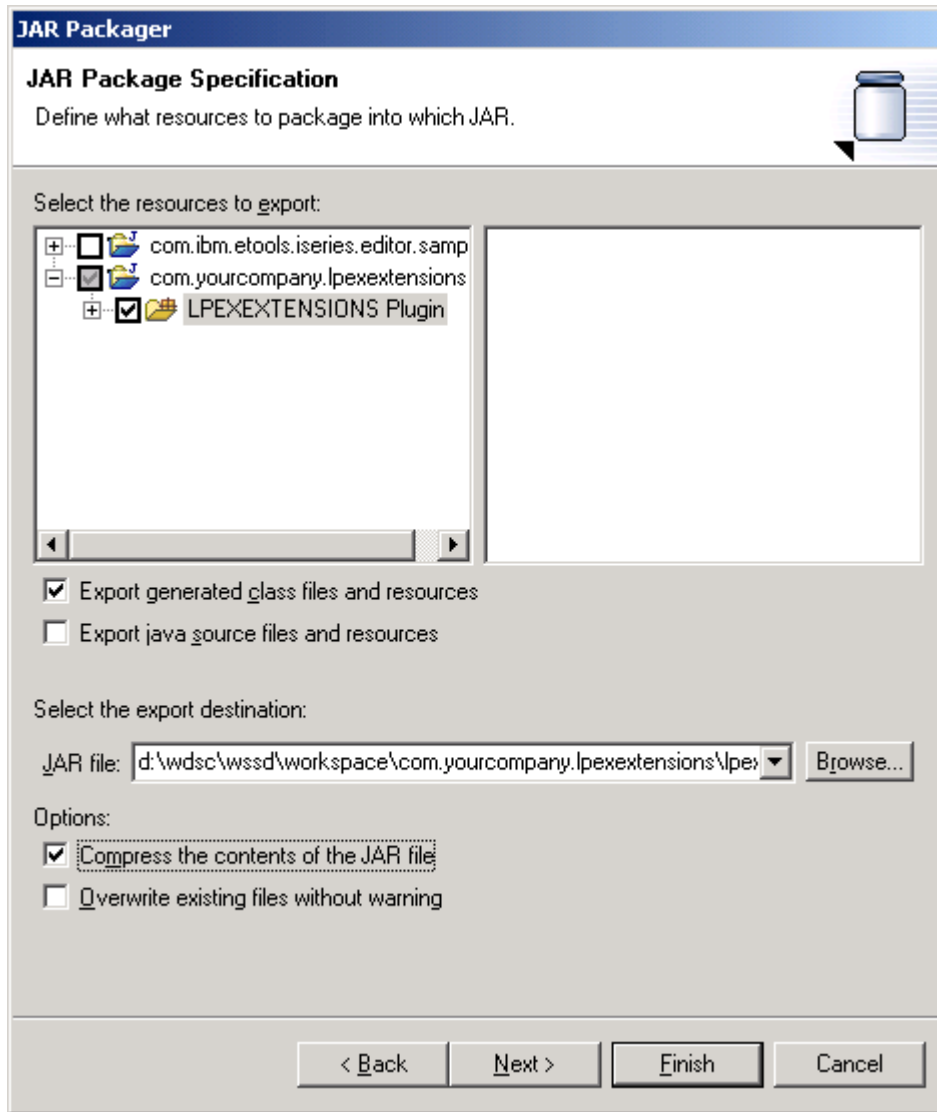
Click Finish to have the JAR file generated. If you change the Java files after packaging, you will need to do this packaging step again. Now to share your plugin with others, they would need to create a directory called 'com.yourcompany.lpexextensions' under their c:\WDSC\WSSD\plugins directory. In that new directory, they would need to place your plugin.xml file and the lpexextensions.jar file. After they restart WDSC, they should be able to use your user commands in the LPEX Editor.

Conclusion:

This just shows a fraction of the capabilities that you have by writing your own extensions to the LPEX Editor. For more information about the LPEX Editor, you can browse the online help. It has full documentation on the default editor commands available as well as the API reference for

people writing their own user commands. In addition to user commands, the LPEX Editor also enables you to write your own user actions and document parsers. As with the CODE Editor, the LPEX Editor offers complete control over your editing experience.