Note!

Before using this information and the product it supports, be sure to read the general information under Notices.
### Contents

**Chapter 1. The IBM Distributed Debugger for AS/400**

- Distributed Debugger: Overview ........................................ 1
- Distributed Debugger: Panes ........................................... 1
- Distributed Debugger: Monitors ....................................... 2
- When to attach ............................................................. 3
- Source views ............................................................... 4
- Secondary threads in the debugger .................................... 4
- Optimized code debugging ............................................... 5
- Search order ............................................................... 5

**Chapter 2. Preparing a program for debugging**

- When you start debugging ............................................... 7
- Writing a program for debugging ...................................... 8
- Compiling a program for debugging ................................... 8
- Starting or stopping the debug server ................................. 9
  - Starting the debug server ........................................... 9
  - Ending the debug server ............................................ 9
- Setting up a debug session ............................................. 9
  - Preparing for debugging ............................................ 9
  - Setting the debugger port ........................................... 10
  - Specifying parameters when starting the host program to debug ........................................... 10
  - Specifying the name of the debug host ......................... 11
- Setting the library list and current library ....................... 11

**Chapter 3. Starting the debugger**

- Loading a program into the debugger without specifying a job ........................................... 16
  - Loading a program without specifying a job from a command line ........................................... 16
  - Loading a program without specifying a job from the debugger user interface .......................... 17
- Attaching to an existing job on an AS/400 .................................. 17
  - Attaching to an existing AS/400 job from a command line ........................................... 17
  - Attaching to an existing AS/400 job from the debugger user interface .................................... 18
- Starting the Distributed Debugger user interface daemon ........................................... 19
- Starting the debugger for a threaded application .................. 20
- Starting the debugger for a non-threaded ILE or OPM application ........................................... 20
- idebug command ............................................................. 21

**Chapter 4. Using the IBM Distributed Debugger for AS/400**

- Setting and working with environment variables ................... 27
- Setting the Java CLASSPATH ........................................... 28
- Working with programs ................................................... 28
  - Adding a program to the Programs pane ......................... 28
  - Using the Programs pane ............................................ 29
- Setting the display of program information .......................... 30
  - Changing views in the Source Pane ................................ 30
  - Viewing different source ........................................... 30
- Locating source code .................................................... 31
  - Locating source code written in C++ .............................. 31
  - Locating source code written in ILE/OPM languages .................. 32
  - Locating source code written in Java .............................. 32
- Viewing the contents of the Call Stack ............................... 33
- Controlling program execution ......................................... 33
- Working with threads .................................................... 36
- Inspecting variables ..................................................... 37
- Working with the debugger user interface ........................... 43
- Breakpoints ............................................................... 44
  - Line breakpoints ....................................................... 45
  - Watch breakpoints .................................................... 45
  - Characteristics of watches ........................................ 46
  - Thread-specific and job-wide line breakpoints ................ 47
  - Setting Breakpoints ................................................ 47
  - Viewing set breakpoints ............................................ 50
  - Modifying line breakpoint properties ............................. 51
  - Enabling and disabling breakpoints ................................ 52
  - Deleting a breakpoint ................................................ 53
- Debugging servlets, EJB, and JSP ...................................... 54
  - Debugging servlets, JSP, and EJB ................................ 54
  - Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP .................. 55
  - Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP .................. 56
  - Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging ........ 56
  - Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging ........ 57
  - Preparing servlets, EJB, and JSP for debugging ................. 61
  - An EJB example ....................................................... 66
  - A servlet example .................................................... 67
  - A JavaServer Page example ......................................... 68
- Reference information ................................................... 73
  - Debug data and RPG, C, Cobol, and CL compiler options ............ 73
  - ILE C++ compiler options ........................................... 74
  - Java compiler options ............................................... 74
  - Creating a Java program from a Java class ....................... 75
  - Debugger environment .............................................. 75
  - Environment variables .............................................. 76
  - Stepping ............................................................... 77
  - Program name specifications ....................................... 78
  - Job name specifications and getting a job list ................... 78
  - Job types ............................................................. 79
  - Authorities required for debugging ................................ 80
  - Debugger limits ....................................................... 80
  - Debugger performance considerations ................................ 81
  - Expressions qualified to thread number ........................... 82
  - Problems getting a Source or Listing view ....................... 82
  - Values that are valid for the current representation .............. 83
  - Previous load program methods ..................................... 83

© Copyright IBM Corp. 1997, 2000
Chapter 1. The IBM Distributed Debugger for AS/400

Distributed Debugger: Overview

The IBM Distributed Debugger is a client/server application that enables you to detect and diagnose errors in code developed with AS/400 Integrated Language Environment (ILE) and Original Program Model (OPM) languages and Java. This client/server design makes it possible to debug applications running on an AS/400 machine while the debugger user interface is running on a Windows workstation.

The debug server runs on the same system as the program you want to debug. When you start the debugger, you have the option of loading a program without specifying a job or attaching to an existing job on the AS/400. When you load a program without specifying a job, the debugger creates a job on the AS/400 for you.

The Distributed Debugger client is a graphical user interface where you can issue commands used by a debug engine to control program execution. For example, you can run your program, set breakpoints, step through program instructions, examine variables, and examine the call stack. The Distributed Debugger user interface lets you debug multiple applications, which may be written in different languages, from a single debugger window. Each program you debug is shown on a separate program page with a tab on each page displaying program identification information such as the name of the program or a job being debugged.

Each program page is divided into sections, called panes. Each pane displays different information about your program. There are panes to display your source code, breakpoints, the program’s call stack, and various monitors.

For more information on the panes and monitors available in the Distributed Debugger user interface, see the related topics below.

RELATED CONCEPTS
Distributed Debugger: Panes
Distributed Debugger: Monitors
When to attach

RELATED TASKS
Preparing for debugging

Distributed Debugger: Panes

The following panes are available in the Distributed Debugger user interface.

Stacks pane
The Stacks pane provides a view of the call stack for each thread in the program you are debugging. Each thread in your program appears as a root node in a tree structure. Expand a node to display the names of the functions on the call stack for that thread.

If the Stacks pane is not in view, select View from the menu bar. The Stacks pane is in view when the Control Panes option in the View menu is checked.
**Breakpoints pane**

The Breakpoints pane provides a view of all breakpoints that you have set for the debug session. **Line Breakpoints** appears as a root node in a tree structure if your program contains line breakpoints and **Watch Breakpoints** appears as a root node if your program contains watch breakpoints. Expanding a node will display a list of the breakpoints of the corresponding type.

Use the Breakpoints pane to view breakpoints that have been set in your program or delete them. You can also use the Breakpoints pane to modify the properties of line breakpoints.

If the Breakpoints pane is not in view, select **View** from the menu bar. The Breakpoints pane is in view when the **Control Panes** option in the **View** menu is checked.

**Programs pane**

The Programs pane displays a list of Programs, Service Programs and Java classes under debug in the current debug session. Each program appears as a root node in a tree structure. You can expand the program nodes to show modules, source files and procedures or functions.

If the Programs pane is not in view, select **View** from the menu bar. The Programs pane is in view when the **Control Panes** option in the **View** menu is checked.

**Source pane**

The Source pane provides various debug views of the program you are debugging. You have up to three choices for viewing your program, if it was compiled with the appropriate debugging information: Source view, Listing view, or Statement view. To display a Source view, the source file must be available on the AS/400 or locally.

For Java classes, the Source and Statement views may be available.

If the Source pane is not in view, select **View** from the menu bar. The Source pane is in view when the **Source Pane** option in the **View** menu is checked.

**RELATED CONCEPTS**

- Distributed Debugger: Monitors
- Distributed Debugger: Overview
- Source views

**RELATED TASKS**

- Changing views in the Source pane
- Viewing the contents of the Call Stack

**RELATED REFERENCES**

- Problems getting a Source or Listing view

**Distributed Debugger: Monitors**

The Distributed Debugger provides you with two monitors to monitor variables in a program. You can select a monitor from the Distributed Debugger user interface.

**Variables and Expressions (Monitors pane)**

The Monitors pane shows variables that you have added to it. You can add
variables to the Monitors pane by opening a dialog box or by selecting them from the Source pane. Use the Monitors pane to monitor global variables or variables you want to see at all times during your debugging session. From the Monitors pane you can also modify the content of variables or change the representation of values.

If the Monitors pane is not in view, select View from the menu bar. The Monitors pane is in view when the Value Panes option in the View menu is checked.

Tip: Enabling Tool Tip Evaluation for variables provides a quick way to view the contents of variables in the Source pane. When you point at a variable, a pop-up appears displaying the contents of that variable. If Tool Tip Evaluation is disabled and you want to enable it, see the related topic below.

Local Variables (Locals pane)
The Locals pane helps you monitor all local variables in scope at the current execution point of your program. For multithreaded programs, each thread is listed and can be expanded to show the local variables in scope for that thread. The Locals pane is updated after each Step or Run command to reflect which variables are currently in scope and the contents of those variables. It can also be used to change the content of variables.

If the Locals pane is not in view, select View from the menu bar. The Locals pane is in view when the Value Panes option in the View menu is checked.

When to attach

You may choose to attach to an existing AS/400 job if you are debugging an application in that specific job. In so doing, you can have complete control over that job and the application in the job can be running or not running.

Another reason you may want to attach to an existing job is that you anticipate a problem at a particular point in your program, and you do not want to step through the program or set breakpoints. In this situation, you can run your program, and at a program pause shortly before the anticipated failure (for example, while the program is waiting for keyboard input), you attach the debugger. You can then provide the input, and debug from that point on.

Finally, attach to an existing job if you are developing or maintaining a program that stops responding sporadically, and you want to find out why it stops responding. In this situation, you can attach the debugger, and look for infinite loops or other problems that might be causing your program to stop.

To attach to an existing AS/400 job, you need to supply the job number.
Starting the debugger
Attaching to an existing AS/400 job from a command line
Attaching to an existing AS/400 job from the debugger user interface

Source views

If your program was compiled with the appropriate debugging information, you have up to three choices about how to view it in the Source pane.

Source View
Select Source > Source View to switch to a Source view of the program being debugged. This view displays the source code for your program.

If the program was compiled with Source debug view, but the source file cannot be located, a dialog box prompts you for the location of the source file on the workstation. If you click Cancel in the dialog box, the debugger will look for the Listing view. (A Listing view is available for OPM or ILE programs that have been compiled with the corresponding compiler option.) If the Listing view is unavailable, the Statement view is displayed.

Note: Whether the Source view is available or not depends on the compiler options that you used to create debug information.

Listing View
Select Source > Listing View to switch to a Listing view of the program being debugged. This view displays a compiler-generated listing of your program. The Listing view includes Copy files.

Note: Whether the Listing view is available or not depends on the compiler options that were used to create debug information.

Statement View
Select Source > Statement to switch to a statement view of the program being debugged. This view displays the statement number for each executable statement, as well as the name of the procedure to which the statement belongs.

Secondary threads in the debugger

You can load a multithreaded application into the debugger without specifying a job or attach to it when it is in an existing job.

A threaded job consists of one job and one or more threads. The initial thread is created during the creation of the job. Other threads, called secondary threads, are created by the application using a thread create API.

On start-up, the debugger will stop in the initial thread at the first executable statement if debug data is available.
The debugger will display the source for the *initial thread* and you will be able to set breakpoints in the initial thread or in functions or procedures that will run in *secondary threads* within this job.

When a thread is stopped because a step is completed, a breakpoint is hit, a watch condition is satisfied, or an unmonitored exception is received, the debugger will halt all other threads running in this job. The debugger displays the stopped thread’s source and highlights the current line of execution for that thread.

Issuing a step or a run command will resume the program. The thread that is displayed in the Source pane will have the highest priority and its next step will be signaled first.

**RELATED TASKS**
- Debugging multithreaded applications
- Starting the debugger for a threaded application
- Using the Stacks pane

### Optimized code debugging

Generally, the higher the optimization level, the more efficiently your program runs. However, if the program you are debugging is optimized, the value of displayed variables may not be current.

While debugging your code, set the optimization level to the minimum level. This allows you to accurately display and change variables. After you have completed your debugging session, set the optimization level to the maximum level. This provides the highest levels of performance for the procedures in the program. See your compiler documentation for details on specifying optimization levels.

**Note:** Even at optimization level *NONE, some optimization may be done in certain cases that could affect the debugger’s ability to accurately display the program’s stopped location.

**RELATED TASKS**
- Writing a program for debugging
- Compiling a program for debugging

**RELATED REFERENCES**
- Debug data and RPG, C, Cobol, and CL compiler options
- ILE C++ compiler options
- Java compiler options

### Search order

The debugger searches for sources for a program to be debugged on the AS/400 in the following locations, depending on the format you use to specify the program name in the *Specify the name of the program to add to debug* entry field of the Load Program dialog box:

<table>
<thead>
<tr>
<th>Library/program</th>
<th>The debugger searches the specified library for the program. If it does not find the program in the specified library, an error message is displayed.</th>
</tr>
</thead>
</table>

Chapter 1. The IBM Distributed Debugger for AS/400
**LIBL/program**  
The debugger searches all of the libraries in the library list for the job specified in the **Job to debug** entry field of the Load Program dialog box until it finds the first match for the specified program name. If it does not find the program in the library list, an error message is displayed.

**CURLIB/program**  
The debugger uses the current library for the job specified in the **Job to debug** entry field of the Load Program dialog box to locate the program. If you do not have a library designed as the current library, it searches QGPL instead. If it does not find the program, an error message is displayed.

**Program**  
When only the program name is specified, the search path used is the same as that used for **LIBL/program**.

If the debugger cannot locate the source in any of the above libraries, a dialog box opens requesting the local path name for the source.

**JAVA**  
For Java programs, the debugger assumes that the source is in the same location as the `{classname}.class` file, in the Integrated File System (IFS), and that it has the same file name, but with the extension `.java`. If the source is not found there (or if you do not have sufficient authority to access the source file), the debugger will search its debug source path (which you can set in the Debugger Settings dialog box). If the file is still not found, a dialog box opens requesting the local path name for the source. If you want to override the source location to another IFS drive, you must first map the network drive and specify the location from the mapped drive, for example, `mapped-drive-letter\path\filename.ext`

**RELATED TASKS**
Locating source code
Locating source code written in C++
Locating source code written in ILE/OPM languages
Locating source code written in Java
Chapter 2. Preparing a program for debugging

When you start debugging

**Restriction:** When you select the File menu from the menu bar, only the **Load Program** option applies when debugging on the AS/400. In the File menu, **Process List** and **Attach to JVM** are not applicable when debugging on the AS/400.

The first time you start the debugger, the main debugger window opens along with the following, depending on how the debugger was invoked:

- The Load Program dialog box opens if the `idebug` invocation command is used without the parameters that are necessary for loading a program without specifying a job or attaching to an existing AS/400 job. For information about loading a program into the debugger without specifying a job or attaching to a job, please see the related topics below.
- If the Debugger AS/400 Logon dialog box opens, specify your AS/400 user ID, password, and TCP/IP address of the AS/400 host.
- If you are loading a program without specifying a job, the debugger starts the program for you and a Source pane appears for the program. Depending on the compiler options used, the Source pane contains either a Source, Listing, or Statement view of the module that contains the entry procedure of your program.
- When attaching to a job, if you select **Step Into**, you will be asked to start your AS/400 application after clicking the **Load** push button. If you do not select **Step Into**, you will be able to view the source, add new programs or classes for debugging, and set breakpoints after clicking the **Load** push button. If you click the **Run** push button in the main debugger window, a message will prompt you to start your AS/400 application. At this point, you will need to invoke the application on the AS/400 if is not already running. Once the application is running, click the message dialog’s **OK** button. The application will stop at the first breakpoint in the code.
- The AS/400 Java Console window, if you are loading a Java class into the debugger without specifying a job.

When you start debugging a program for the first time, no breakpoints are set and no variables are in the Monitors pane. During the debug session, you may set breakpoints, and add variables or expressions to a monitor. When you exit the debugger, these breakpoints, variables and expressions are saved in the program profile. The next time you start debugging the same program, these settings will apply if you select **Use program profile** in the Load Program dialog box.

**Related Tasks**
- Preparing for debugging
- Starting the debugger
- Loading a program without specifying a job from a command line
- Loading a program without specifying a job from the debugger user interface
- Attaching to an existing AS/400 job from a command line
- Attaching to an existing AS/400 job from the debugger user interface

**Related References**
- `idebug` command parameters and options
Writing a program for debugging

To make your programs easier to debug, follow these guidelines:

- Avoid putting multiple statements on a single line. Some Distributed Debugger features operate on a line basis. For example, you cannot step over or set line breakpoints on more than one statement on the same line.
- Assign intermediate expression values to temporary variables so you can verify intermediate results by monitoring the temporary variables.
- If you have a bug in a complex expression, you can debug the expression more easily by breaking it down into intermediate values and by assigning those values to temporary variables. This allows you to verify each expression and variable. For example, you will not be able to display the substrings of IString objects in the first C++ code fragment below, but you will in the second:

```cpp
// Can't see the substrings in this one
if (StrA.subString(x,y)==StrB.subString(m,n)) dups++;
// Can see the substrings here
IString SubA=StrA.subString(x,y);
IString SubB=StrB.subString(m,n)
if (SubA==SubB) dups++;
```

- To debug programs at the level of source code statements, you must specify the compiler options that generate debug information. In some cases, you must specify additional options that enable the debug engine to work properly with your code.

**Related Concepts**
Optimized code debugging

**Related Tasks**
Compiling a program for debugging

**Related References**
Debug data and RPG, C, Cobol, and CL compiler options
ILE C++ compiler options
Java compiler options

Compiling a program for debugging

In order to debug your program at the source code level, you must set certain compiler options to generate symbolic information and debug hooks in the object file. For more information about compiling each language, see the related references below:

<table>
<thead>
<tr>
<th>Language</th>
<th>Compile using...</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPM COBOL or RPG</td>
<td>either the *SRCDBG or *LSTDBG option.</td>
</tr>
<tr>
<td>OPM CL</td>
<td>the *SRCDBG option.</td>
</tr>
<tr>
<td>ILE COBOL, CL, or RPG</td>
<td>the *SOURCE, *LIST, *STMT, or *ALL option. The default is *STMT (Statement view).</td>
</tr>
<tr>
<td>ILE C</td>
<td>the *SOURCE, *LIST, *STMT, or *ALL option. The default is *NONE (no debug data).</td>
</tr>
</tbody>
</table>

See your compiler documentation for more details on these compiler options.
Starting or stopping the debug server

Starting the debug server

Before you can use the debugger, the debug server must be started on the AS/400. Type the Start Debug Server (STRDBGSVR) command on an AS/400 command line and press Enter.

Note: The debug server needs to be started only once, for the AS/400 system on which you are planning to debug your application. If the debug server has already been started previously, you will see the message “Debug server router function already active”, when you issue the STRDBGSVR command.

Ending the debug server

Normally, you do not end the debug server at the end of your debug session, because the debugger is active for the entire AS/400 system.

If necessary, end the debug server using the End Debug Server (ENDDBGSVR) command on an AS/400 command line.

This prevents any new debug sessions from starting.

Setting up a debug session

Preparing for debugging

Before you can start debugging an AS/400 application, you may need to perform some or all of the following tasks:

- Compile a program with debug data.
- Start the debug server.
- Start a debug session.
• Set the debugger port.
• Change the debugger settings.
• Ensure that your AS/400 user profile has the appropriate authorities.

For information about these tasks, see the related information below.

RELATED CONCEPTS
Distributed Debugger: Overview

RELATED TASKS
Compiling a program for debugging
Starting the debug server
Setting the debugger port

RELATED REFERENCES
Authorities required for debugging
Debugger environment settings

Setting the debugger port

By default, the debug server is set up to listen for connection requests on TCP/IP port 3001. If port 3001 is being used by another application, you can designate another port:

1. Use the Work Server Table Entry (WRKSRVTBLE) command on the AS/400 debug host and change the port called QDBGSVR to the new value. **Note:** Before changing the port, end the debug server by using the End Debug Server (ENDDBGSVR) command. Then change the port and start the debug server again.

2. To sign on to the AS/400 debug host for the first time, specify the new port in the **Host Name** entry field of the Debugger AS/400 Logon dialog box as hostname:portnumber, where hostname is the name of the AS/400 system on which you want to run the application to be debugged, and portnumber is the new value of the QDBGSVR port.

3. When the Load Program dialog box appears, press the **Advanced** push button.

4. Change the **Port Number** setting in the **Session Settings** group of the Debugger Settings dialog box to match the port specified on the AS/400 WRKSRVTBLE command.

RELATED TASKS
Starting the debugger
Loading a program without specifying a job from the debugger user interface
Attaching to an existing AS/400 job from the debugger user interface
Specifying the name of the debug host
Terminating a debug session

RELATED REFERENCES
idebug command parameters and options

Specifying parameters when starting the host program to debug

When you load a program without specifying a job name, the debugger automatically creates a job and invokes the program in that job. You need to supply all parameters required to invoke the program, in the **Enter any program parameters** field of the Load Program dialog box.
When you attach the debugger to an existing AS/400 job, specify program parameters from the AS/400 command line when you start the program on the AS/400. To pass parameters to your program from an AS/400 command line, use the PARM option of the CL CALL command. The syntax for this command is:

```
CALL PGM(program-name) PARM(program-parameters)
```

where `program-name` is the name of the program that you want to debug, and `program-parameters` are the arguments or parameters you want to pass to your program.

You can also type the parameters without specifying any keywords:

```
CALL library/program-name (program-parameters)
```

**Related Tasks**
- Preparing for debugging
- Starting the debugger

### Specifying the name of the debug host

You can specify the name of the debug host to which you want to connect in one of the following ways:

- Through the debugger invocation parameter `-e<hostname:port>`, where specifying port is optional.
- In the `Session Settings` group of the Debugger Settings dialog box.
- In the Debugger AS/400 Logon dialog box.

Note the following precedence rules:

- If you invoke the debugger with the `idebug -qengine=400 -e<hostname:port>` command, `hostname:port` is used to connect to the debug host.
- If you invoke the debugger from the debugger icon or from the command line without the `-e<hostname:port>` parameter, the hostname saved in the Debugger Settings dialog box will be used. If you have not saved a hostname, you must specify the name of the debug host in the Host Name entry field of the Debugger AS/400 Logon dialog box.

**Related Tasks**
- Starting the debugger
- Loading a program without specifying a job from the debugger user interface
- Attaching to an existing AS/400 job from the debugger user interface
- Setting the debugger port
- Terminating a debug session

**Related References**
- `idebug` command parameters and options

### Setting the library list and current library

Set the library list and current library to specify the libraries the debugger should search to locate programs on the AS/400 when called from a Java application or a batch immediate (BCI) job. The current library is the first library to be used when debugging. Library settings are made in the Debugger Settings dialog box.

The Debugger Settings dialog box will not close if you specify incorrect values in it and click the **OK** push button. For example, the dialog box will not close if you enter a library name that is longer than 10 characters or contains blanks. All fields
which contain an error will be marked in red. Tool Tip Evaluation will give details when you position the cursor over the field containing the entry.

When debugging Java, the JNI library list and current library are set when specifying JVM creation settings. The JNI library list and current library are useful when debugging Java applications that make JNI calls.

Setting the library list
To view the library list, do the following:
1. Select File > Load Program from the menu bar to invoke the Load Program dialog box.
2. In the Load Program dialog box, click the Advanced push button to invoke the Debugger Settings dialog box.
3. In the Debugger Settings dialog box, do one of the following:
   - If you are debugging a Java application, select JVM Creation Settings > JNI Library List.
   - If you are debugging a batch immediate (BCI) job, select Batch Job Creation Settings > Library List.
4. The Library List in the Debugger Settings dialog box will contain the first 20 AS/400 libraries that were specified the last time you debugged an application. If no libraries were specified, the list will contain no libraries by default.

Entries in the Library List in the Debugger Settings dialog box can be added, modified, or deleted as follows:
   - To add a first library in the library list, click on the first Library List cell field and make the appropriate entries. To add additional libraries, press the keyboard Insert key to create rows for as many libraries as you wish to add and make the appropriate entries.
   - To edit library names, click on the appropriate table cell if it is highlighted or double-click on the appropriate cell if it is not highlighted, and change the cell contents.
   - To delete a library from the library list, select the appropriate row, and press the keyboard Delete key.

Entries in the library list table can be reordered by dragging and dropping rows.

When the Library List has been set as desired, click OK to set the list and dismiss the Debugger Settings dialog box.

Setting the current library
To set the current library, do the following:
1. Select File > Load Program from the menu bar to invoke the Load Program dialog box.
2. In the Load Program dialog box, click the Advanced push button to invoke the Debugger Settings dialog box.
3. In the Debugger Settings dialog box, do one of the following:
   - If you are debugging a Java application, select JVM Creation Settings > JNI Library List.
   - If you are debugging a batch immediate (BCI) job, select Batch Job Creation Settings > Library List.
4. In the Current Library entry field, enter the name of the library.
5. Click OK to set the current library and dismiss the Debugger Settings dialog box.
RELATED CONCEPTS
Search order

RELATED TASKS
Starting the debugger
Locating source code
Setting and working with environment variables
Chapter 3. Starting the debugger

You can invoke the debugger from a command line, from VisualAge for Java Enterprise Edition, from the CODE Editor, from the CODE Project Organizer, or by clicking the debugger icon.

Starting the debugger from a command line
To start the debugger from a workstation command line, use the idebug invocation command. Depending on options and parameters used with this command, you can load a program for debugging without specifying a job or attach to an existing job on an AS/400. For information about loading a program for debugging from a command line or attaching to an existing AS/400 job from a command line, see the related tasks below. For information about idebug options and parameters, see the related reference below.

If you do not specify idebug options or parameters that are necessary to attach to a job or load a program without specifying a job, you will be required to make the necessary entries in the Load Program dialog box, once the debugger user interface is running.

You can also debug other programs using the same debugger session by selecting File > Load Program from the debugger user interface menu bar.

Starting the debugger from VisualAge for Java Enterprise Edition
When you wish to debug Java programs other than servlets, JavaServer Pages (JSP), and enterprise beans (EJB), start the debugger from VisualAge for Java Enterprise Edition by doing the following:
1. Ensure that the debug server is running on the AS/400.
2. From the VisualAge for Java Workbench, select the project, package, or class that you wish to debug.
3. Right click and select Tools then ET/400 then Properties.
4. In the AS/400 Properties dialog box, specify options to Export, Compile, and Run if they are not already set. In Export Options, ensure that both Debuggable Classes and Both Java and Class Files are selected. Click OK to close the AS/400 Properties dialog box.
5. Right click and select Tools then ET/400 then Export to export the class and source files to an AS/400 IFS directory, if the Java application you want to debug is not already on the AS/400.
6. Right click and select Tools then ET/400 then Debug to start the debugger.
7. The AS/400 Java Console window and a Source view window open.
8. You can now start debugging your program.

Using VisualAge for Java Enterprise Edition for servlet, JavaServer Page, and enterprise bean debugging
To debug servlets, JavaServer Pages (JSP), and enterprise beans (EJB), please refer to the related concept below. One step involved in debugging these web artifacts is starting Object Level Trace (OLT) Viewer. You can start OLT Viewer by doing the following in the Workbench:
1. Ensure that you are using JDK 1.2.2.
2. Ensure that the IBM Distributed Debugger for AS/400 has been installed and that, upon its installation, Object Level Trace was installed.
3. In the VisualAge for Java Workbench, right click and select **Tools > ET/400 > Start OLT viewer** from the pop-up menu.

**RELATED CONCEPTS**
Distributed Debugger: Overview
Debugging servlets, JSP, and EJB

**RELATED TASKS**
Loading a program without specifying a job from a command line
Loading a program without specifying a job from the debugger user interface
Attaching to an existing AS/400 job from a command line
Attaching to an existing AS/400 job from the debugger user interface
Specifying the name of the debug host
Terminating a debug session

**RELATED REFERENCES**
idebug command parameters and options

---

## Loading a program into the debugger without specifying a job

### Loading a program without specifying a job from a command line

To start the debugger from a workstation command line and load a program for debugging without specifying a job, do the following:

1. Ensure that the debug server is running on the AS/400.
2. Enter the debugger invocation command and parameters
   ```
   idebug -qengine=400 program_name
   ```
3. To use additional options and parameters with this command, type:
   ```
   idebug -qengine=400 program_name x
   ```
   where `x` represents any number of optional debugger invocation parameters.

If the above parameters are not specified, the debugger will prompt you for the required information in the Load Program dialog box. For information about loading a program without specifying a job from the debugger user interface, see the related task below.

The presence of a dominant language is also mandatory when loading a program, however, if the `-qlang=<dominant_language>` parameter is not specified, the dominant language last used by the debugger will be assumed for the current session.

For information about optional debugger invocation parameters, please see the related reference below.

**RELATED CONCEPTS**
Distributed Debugger: Overview

**RELATED TASKS**
Starting the debugger
Attaching to an existing AS/400 job from a command line
Specifying the name of the debug host
Terminating a debug session
Loading a program without specifying a job from the debugger user interface

You can load a program for debugging without specifying a job during startup.

To load a program from the debugger user interface without specifying a job, do the following:

1. Select File > Load Program from the menu bar. The Load Program dialog box appears.
2. In the Load Program dialog box, enter the name of the program you want to start for the debug session in the Specify the name of the program to add to debug entry field.
3. Enter optional debugger invocation parameters in the Enter any program parameters entry field. For more information on optional debugger invocation parameters, please see the related reference below.
4. If the dominant language to debug was not specified as a parameter when invoking the debugger, enter the language in the program language field, or choose it from the list of language options in the field’s pull-down list.
5. Enter optional debugger session settings, JVM creation settings, or batch job creation settings in the Debugger Settings dialog box. The Debugger Settings dialog box appears when you click on the Advanced push button in the Load Program dialog box.
6. Select the Load push button to load the specified program.

Attaching to an existing job on an AS/400

Attaching to an existing AS/400 job from a command line

To start the debugger from a workstation command line and attach to a job that exists on an AS/400, do the following:

1. Ensure that the debug server and job that you are attaching to are running on the AS/400.
2. Enter the debugger invocation command and parameters
   
   idebug -qengine=400 -ajob_name

3. To use additional options and parameters with this command, type:
   
   idebug -qengine=400 -ajob_name x
   
   where x represents any number of optional debugger invocation parameters.
If the above parameters are not specified, the debugger will prompt you for the required information in the Load Program dialog box. For more information about attaching to an existing AS/400 job from the debugger user interface, see the related task below.

The presence of a dominant language is also mandatory when attaching to an existing job, however, if the -qlang=dominant_language parameter is not specified, the dominant language last used by the debugger will be assumed for the current session.

For information about optional debugger invocation parameters, please see the related reference below.

**RELATED CONCEPTS**
Distributed Debugger: Overview

**RELATED TASKS**
Starting the debugger
Attaching to an existing AS/400 job from the debugger user interface
Loading a program without specifying a job from a command line
Specifying the name of the debug host
Terminating a debug session

**RELATED REFERENCES**
idebug command parameters and options

**Attaching to an existing AS/400 job from the debugger user interface**

If you are attaching to a program that was written in C/C++, Cobol, CL, or RPG, then you are attaching to a job.

If you are attaching to a process that was written in Java, then you are attaching to a running Java Virtual Machine (JVM).

When you start the debugger, you can attach it to a process or JVM that exists on an AS/400 by specifying the job name associated with the process.

To attach to an existing AS/400 job from the debugger user interface, do the following:

1. Select File > Load Program from the menu bar. The Load Program dialog box appears.
2. In the Job to debug field, enter the job name that is associated with the process that you wish to attach to. Alternatively, click the Job List push button to invoke the Job List dialog box from which you can select the appropriate job which is associated with the process you wish to attach to.
3. Entering the name of the program to debug in the program entry field is optional.
   - If the program name is specified in this field, the program will be added as an entry to the Programs pane. If the Step Into box is also checked, the debugger will stop at the first debuggable statement which runs in the job you specified.
• If the program name is specified, running the program to termination will result in debug session termination and invocation of the Load Program dialog box for a subsequent debug session.

• If the program name is not specified, running the program to termination will leave the debugger client waiting for a response from the server. In this case, the debugger must be manually terminated or restarted for additional debug sessions.

4. Enter optional debugger invocation parameters in the program parameters entry field. For more information on optional debugger invocation parameters, please see the related reference below.

5. If the dominant language to debug was not specified as a parameter when invoking the debugger, enter the language in the program language field, or choose it from the list of language options in the field’s pull-down list.

6. Enter optional debugger session settings, JVM creation settings, or batch job creation settings in the Debugger Settings dialog box. The Debugger Settings dialog box appears when you click on the Advanced push button in the Load Program dialog box.

7. If Step Into in the Select startup behavior control group is selected, you will be asked to start your AS/400 application after clicking the Load push button. If it is not selected, you will be asked to start your application only when attempting to run the program in the debugger.

8. Select the Load push button to attach to the specified job.

**RELATED CONCEPTS**
When to attach

**RELATED TASKS**
Starting the debugger
Attaching to an existing AS/400 job from a command line
Running a program

**RELATED REFERENCES**
idebug command parameters and options
Debugger environment settings
Job types
Job name specifications and getting a job list
Program name specifications

---

**Starting the Distributed Debugger user interface daemon**

Start the Distributed Debugger user interface in daemon mode if you want the Distributed Debugger user interface to appear only after you have started a debug engine.

To start the Distributed Debugger user interface daemon, issue the following command at a command line prompt:

`idebug -qdaemon -quiport=<port>`

where `<port>` is the port number where you want the Distributed Debugger user interface daemon to listen for a debug engine.

**RELATED TASKS**
Starting the debugger
Starting the debugger for a threaded application

**Note:** A thread is an independent unit of execution within an AS/400 job. A job is thread-enabled if it is spawned as an immediate batch job, with support for native threads.

To debug a threaded application, in the Load Program dialog box specify the name of a threaded application in the **Specify the name of the program to add to debug** entry field. Since no job name is specified, the debugger will spawn a job and start the program in that job.

**RELATED CONCEPTS**
Secondary threads in the debugger

**RELATED TASKS**
Starting the debugger

Starting the debugger for a non-threaded ILE or OPM application

To debug a non-threaded ILE or OPM program, proceed as follows:
1. Specify the name of the non-threaded ILE or OPM program in the program name entry field of the Load Program dialog box.
2. Type the name of your interactive AS/400 job in the **Job to debug** entry field. For a list of jobs, select the **Job List** push button and select a job from the Job List dialog box.
3. Select the **Advanced** push button in the Load Program dialog box to access the Debugger Settings dialog box.
4. In the Debugger Settings dialog box, specify any environment variables to be processed for the application.
5. Exit the Debugger Settings dialog box.
6. Press the OK push button in the Load Program dialog box. A debugger message appears that asks you to start the program to be debugged on an AS/400.
7. Start the program on the AS/400 debug host.
   **Note:** Program parameters for ILE and OPM programs are specified on the AS/400 command.
8. Press the OK button in the debugger message window to continue. A Source pane opens, displaying one of the debugging views.

You can now start debugging your program.

**RELATED TASKS**
Starting the debugger
Starting the debugger for a threaded application
idebug command

The idebug command connects the Distributed Debugger user interface to the application to be debugged on an AS/400. The idebug command has the following syntax:

`idebug [idebug_options] [ui_daemon_parameters]`

The idebug_options are zero or more of the following:

<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a &lt;job_name&gt;</td>
<td>Attach to the existing AS/400 job_name, where job_name is a string that specifies an AS/400 job. The job_name string follows standard job name format, Name/User/Number. This option does not have a default value.</td>
</tr>
<tr>
<td>-h or -?</td>
<td>Display help for the idebug command.</td>
</tr>
<tr>
<td>-i</td>
<td>Step into the program, but do not run to main. This option has the same function as the Step into check box in the Load Program dialog box.</td>
</tr>
<tr>
<td>Option</td>
<td>Purpose</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
</tr>
</tbody>
</table>
| **-p+** | Use program profile information to keep profiles for initial programs only. The debugger will restore the following for your program from the last time you debugged it:  
  * window sizes, positions, fonts, and breakpoints for your program.  
  * all job breakpoints.  
  * all watch breakpoints will be restored, but disabled.  
  * for single thread programs, variables in the monitor are restored.  
  * for multithreaded programs, variables in the monitor for the initial thread are restored.  
  
The debugger will not restore the following for your program from the last time you debugged it:  
  * thread-specific breakpoints.  
  * thread-specific variables.  
  
If you are debugging a program for the first time, the debugger starts with the default appearance, no breakpoints are set, and no variables are monitored.  
  
Any changes you make to the windows and breakpoints are saved.  

**Note:** If you add or delete lines in your source file, recompile it, and then debug the program again with a saved program profile, line breakpoints may no longer match the code they were initially set for because line breakpoint information is saved by line number, not by the content of the line.  

If the debugger has saved a profile containing information on window, breakpoint, and monitor settings from a previous debug session for this program, the profile is used to restore those settings.  
  
This is the default setting for the debugger. |
<p>| <strong>-p-</strong> | Do not use program profile information. The debugger ignores any program profile information, and the debugger windows start up with their default appearance, and no breakpoints are initially set. |
| <strong>-s</strong> | Prevents the debugger from stopping in the first debuggable statement in the program. Program execution only stops when the first set breakpoint is encountered. |</p>
<table>
<thead>
<tr>
<th>Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-qtitle=&lt;title&gt;</td>
<td>Specifies the title that appears in the process tab for the program you are debugging. The maximum length of the title is 256 characters.</td>
</tr>
</tbody>
</table>

The `ui_daemon_parameters` are used when you start the Distributed Debugger user interface as a daemon. When running as a daemon, the Distributed Debugger user interface listens on a specific port number for a debug engine. Once a connection is made, the Distributed Debugger user interface appears and you can begin debugging your program. The `ui_daemon_parameters` are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-qdaemon</td>
<td>Tells the Distributed Debugger user interface to run as a daemon. You must use the -quiport option when specifying -qdaemon.</td>
</tr>
<tr>
<td>-quiport=&lt;port&gt;</td>
<td>Specifies the port numbers where the Distributed Debugger user interface daemon should listen for a debug engine. You can specify a single port or multiple ports. When specifying multiple ports, <code>&lt;port&gt;</code> must be a comma-delimited list of port numbers. This option is required when using the -qdaemon parameter. When -qdaemon is not used, the debugger does not run as a daemon and the port number defaults to 3001.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| -qlang= `<dominant_language>` | Specifies the dominant language to use for debugging. If this parameter is not specified, the dominant language last used by the debugger will be assumed for the current session.  
Issuing the `idebug -qengine=400` command without this parameter will cause invocation of the Load Program dialog box. Issuing the `idebug -qengine=400` command with this parameter, and with the `program_name` or `-a job_name`, will cause bypass of the Load Program dialog box.  
The initial default value of this parameter is RPG. Thereafter, when a new dominant language is set, it becomes default for subsequent invocations of the Distributed Debugger.  
These are the valid values for `<dominant_language>`:  
- Use the `rpg` value when you are debugging RPG programs.  
- Use the `cobol` value when you are debugging Cobol programs.  
- Use the `cl` value when you are debugging CL programs.  
- Use the `c` value when you are debugging C programs.  
- Use the `cpp` value when you are debugging C++ programs.  
- Use the `java` value when you are debugging Java programs. |
| -qengine=`<engine>` | Specifies the type of engine to connect to. To start the Distributed Debugger, `qengine=400` is a required parameter. |
| -e `hostname:port` | Specifies the name of the AS/400 host and optional server port number that you want to connect to for your debugging session, where specifying the `port` is optional.  
This parameter has no default value. |
| -qprompt | Forces the Load Program dialog box even if all required parameters have been specified. |

When you are attaching to an existing AS/400 job by issuing the `idebug` command, specifying `-a job_name` and `-qengine=`<engine>` are mandatory. When you load a program without specifying a job with the `idebug` command, specifying `-qengine=`<engine>` and `program_name` are mandatory. The presence of a dominant language is also mandatory when attaching to a job or loading a program, however, if `-qlang=`<dominant_language>` is not specified, the dominant language last used by the debugger will be assumed for the current session.
Whether you are attaching to a job or loading a program without specifying a job, if the mandatory options or parameters are not specified, the debugger will prompt you for the required information in the Load Program dialog box.

RELATED TASKS
Starting the debugger
Loading a program without specifying a job from a command line
Loading a program without specifying a job from the debugger user interface
Attaching to an existing AS/400 job from a command line
Attaching to an existing AS/400 job from the debugger user interface
Starting the Distributed Debugger user interface daemon
Starting the debugger for a threaded application
Starting the debugger for a non-threaded ILE or OPM application
Setting and working with environment variables

The Distributed Debugger user interface running on the workstation uses certain environment variables to determine the dominant language, the host files where the source files are found, and so on.

The dominant language to be used by the Distributed Debugger is selected in the Load Program dialog box. To set additional environment variables, select the Advanced push button in the Load Program dialog box. This invokes the Debugger Settings dialog box, where you can set session settings and environment variables such as host name, source path, and debugger port, or add JVM or batch job creation settings.

The Debugger Settings dialog box will not close if you specify incorrect values in it and click the OK push button. For example, the dialog box will not close if you enter an environment variable name that contains blanks. All fields which contain an error will be marked in red. Tool Tip Evaluation will give details when you position the cursor over the field containing the entry.

To set or alter environment variables other than Java CLASSPATH in the Debugger Settings dialog box, do the following:

1. Do one of the following:
   - Select JVM Creation Settings > Environment Variables if you are debugging a job in a Java Virtual Machine.
   - Select Batch Job Creation Settings > Environment Variables if you are debugging a batch immediate (BCI) job.

2. To add an environment variable, click on the Variable and Value cell fields of the initial highlighted row and make the appropriate entries. To add additional variables, press the keyboard Insert key to create rows for as many variables as you wish to add.

3. To edit variable names or values, click on the appropriate table cell and change the cell contents.

4. To delete a variable and its value, select the appropriate row, and press the keyboard Delete key.

Entries in the environment variables table can be reordered by dragging and dropping rows.

Notes:

- AS/400 environment variable names and values may not contain blanks. The environment variable value may not be an empty string.
- Multiple JVM creation environment variable entries of the same name are permitted. The last one in the list is used.
- Multiple JVM creation property entries of the same name are permitted. The first one in the list is used.
Setting the Java CLASSPATH

Set the CLASSPATH environment variable to point at all classes and packages you will need to debug.

To set the Java CLASSPATH, do the following:
1. Select File > Load Program from the menu bar to invoke the Load Program dialog box.
2. In the Load Program dialog box, click the Advanced push button to invoke the Debugger Settings dialog box.
3. In the Debugger Settings dialog box, select JVM Creation Settings.
4. Enter the Java CLASSPATH in the Classpath entry field.
5. Click on the OK button to set the CLASSPATH.

Note: Text in the Classpath field wraps for extra convenience. Do not press Enter while in this field.

Related References
- JAVA CLASSPATH environment variable
- Debugger environment settings
- Debugger environment default settings

Working with programs

Adding a program to the Programs pane

You can add programs, service programs or java classes to the list in the Programs pane of programs under debug mode. Programs under debug mode must include at least one module that has been compiled with debug data. Java classes must be compiled with debug data.

To add a program to the Programs pane from the Programs menu:
1. Click on the Programs pane to bring it into focus.
2. Select Programs > Add Program from the menu bar to invoke the Add Program dialog box.
3. Enter the name of the program, service program, or Java class that you want to add in the Program Name field. Valid formats are:
   • For programs or service programs:
     – Library/program
     – *LIBL/program
     – *CURLIB/program
     – Program (same as *LIBL/program)
   • For Java classes:
     – Full path/Java class
     – Partial path/Java class (if the partial path is in the CLASSPATH as defined in the Debugger Settings dialog box)
4. Specify the type of program that you want to add by selecting one of the types in the Program Type group.

5. Click OK to add the program to the Programs pane and dismiss the dialog box.

Note: Adding a program that has multiple modules with the same name is not supported and will result in an error message.

**RELATED CONCEPTS**
Distributed Debugger: Panes

**RELATED TASKS**
Using the Programs pane

**Using the Programs pane**

Use the Programs pane to view and navigate compilation units such as programs and service programs. Each program or service program display tree can be expanded to display the names of its debuggable modules. Non-debuggable modules are also shown if Show All Components in the Programs pull-down menu is selected. Each debuggable module can be expanded to display the names of all debuggable entries, such as procedures or functions.

When debugging Java, the Programs pane is used to view and navigate Java classes. Each class (or compilation unit) can then be expanded to display names of all debuggable methods that constitute the class.

The Programs pane contains compilation units that are being debugged. Any compilation unit where execution stops automatically gets added to the Programs pane. You can use the Add Program dialog box to add these compilation units to debug. Initially, compilation unit display trees are shown for only those components containing debug information. To display all program components used by the program being debugged, check the Show All Components selection in the Programs pull-down menu.

When a module or Java class is selected, its appropriate view is displayed in the Source pane. When a procedure, function or method is selected, its source is displayed in the Source pane. For more information about viewing different source that is associated with program or class components, see the related topic below.

Compilation unit, module, procedure, and method properties are available by pop-up menu in the Programs pane.

**RELATED CONCEPTS**
Distributed Debugger: Panes

**RELATED TASKS**
Adding a program to the Programs pane
Viewing different source
Setting the display of program information

Changing views in the Source Pane

Depending on source code language and compile options, the Source Pane supports up to three views for programs compiled with debug information:

<table>
<thead>
<tr>
<th>View</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>to see the source code for your program</td>
</tr>
<tr>
<td>Listing</td>
<td>to see a compiler-generated listing of your program</td>
</tr>
<tr>
<td>Statement</td>
<td>to see the procedure name of each executable source statement</td>
</tr>
</tbody>
</table>

You can go directly to a particular view by selecting that view from the Source pull-down menu. For example, to switch to a Listing view, select Source > Listing View.

**RELATED CONCEPTS**
Source views

**RELATED TASKS**
Compiling a program for debugging

**RELATED REFERENCES**
Problems getting a Source or Listing view
Debug data and RPG, C, Cobol, and CL compiler options
ILE C++ compiler options
Java compiler options

Viewing different source

When you start the debugger, a single Source pane appears. Depending on how your program was compiled, the Source pane displays either a Source, Listing, or Statement view.

To open different source from the Source menu, do the following:

1. Select **Source > Open New Module** from the menu bar.
2. Enter the name and path of the module of the new source to be opened. This source code must be a part of the application that you are currently debugging.
3. Click **OK**. A Source pane for the new code opens, if the debugger is able to locate the source.

To open different source from the Programs pane, do the following:

1. Click on the Programs pane to bring it in focus.
2. If necessary, add the program for the source that you wish to view to the Programs pane. For information about adding a program to the Programs pane, see the related topic below.
3. In the Programs pane, expand the entry that contains the program component that you want to view. Alternatively, check the **Show All Components** selection in the Programs pull-down menu.
4. If necessary, scroll the Programs pane to ensure that the entry you want to view is visible. Scroll by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the line.

5. Left-click on the entry you want to view.

6. The source for the selected entry in the Programs pane is displayed in the Source pane.

Viewing different source is useful when you want to set breakpoints in the source of a different program component. When execution switches from the previous source to this one, the program component will stop before running to completion.

**RELATED REFERENCES**

Problems getting a Source or Listing view

**RELATED TASKS**

Adding a program to the Programs pane
Using the Programs pane

---

**Locating source code**

If the debugger cannot locate the source file of the program you are debugging, you may not be able to open a Source view for the module, even if it was compiled with the appropriate debug options.

When you start debugging such a module, or when execution stops in a part of the program that was compiled with the appropriate debug information, but the debugger cannot locate the source file, the Source Filename dialog box allows you to specify the location of the source file.

In this dialog box, you can enter the location and name of the source file. If you choose **Cancel** when this dialog box appears, the Listing view will open if it is available. If the Listing view is unavailable, a Statement view will open.

**RELATED TASKS**

Locating source code written in C++
Locating source code written in ILE/OPM languages
Locating source code written in Java

**Locating source code written in C++**

When you compile C++ source code with the /Ti+ or the /Tis+ option, the compiler stores the name of the source file and its directory path in the module object.

When you ask to display the source file, the debugger tries to find the file, using the directory path and filename that was captured when the program was compiled. The debugger searches for the file in this order:

1. In the directory path that was stored when the module was compiled.
2. In the directory where the last file, if any, was found.
3. In the directories defined in the Source Path environment setting.

If the file is not found in any of these directories, you will be prompted for the name of the file in the Source Filename dialog box. If the source file is not available, click on the **Cancel** push button on the file prompt and a different view will be used.
RELATED TASKS
Locating source code
Locating source code written in ILE/OPM languages
Locating source code written in Java

Locating source code written in ILE/OPM languages

Restriction: The following applies to local source code written in AS/400 languages other than C++ or Java.

When you compile ILE (C, CL, COBOL, and RPG) modules, or OPM (CL, COBOL, RPG) programs, the name of the AS/400 source member is stored in the module or program object. When you ask to display the source for a module or program, the debugger attempts to read the information from the AS/400 source member used to create the module or program.

If the source member is not found, the debugger builds a name using the module name or program name and a file extension based on the language. The file extensions used are as follows:

<table>
<thead>
<tr>
<th>File Extension</th>
<th>Used for...</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>ILE C language modules.</td>
</tr>
<tr>
<td>CBL</td>
<td>ILE COBOL language modules.</td>
</tr>
<tr>
<td>CL</td>
<td>ILE CL language modules.</td>
</tr>
<tr>
<td>RPG</td>
<td>ILE RPG language modules.</td>
</tr>
</tbody>
</table>

The debugger searches for the file in this order:
1. In the directory where the last file, if any, was found.
2. In the directories listed in the Source Path environment setting.

If the file is not found, you are prompted to enter the name of the file in the Source Filename dialog box. If you do not have the source available, click on the Cancel push button on the file prompt and a different view will be used.

RELATED TASKS
Locating source code
Locating source code written in C++
Locating source code written in Java

Locating source code written in Java

When you ask to display the source for a Java class, the debugger tries to locate the source as follows:
1. It searches for a file that has the same IFS location and filename as the class file, but with the extension .java.
2. It searches the path (remote or local) specified in the Source Path environment setting.
3. It prompts you for the name of the file in the Source Filename dialog box.

If you want to override the source location to another IFS drive, you must first map the network drive to your PC, and specify the location from the view of the PC, for example:

mapped-drive-letter:/path/filename.ext
If the source file is not available, press the Cancel push button on the file prompt and a statement view will be displayed.

**RELATED TASKS**
Locating source code
Locating source code written in C++
Locating source code written in ILE/OPM languages

**Viewing the contents of the Call Stack**

You can view information for the active programs, modules, procedures, or functions on a thread’s call stack.

To view stack information, click on the Stacks pane to bring it in focus and expand the appropriate thread entry. Right-click on the call stack entry that you want to view and select **Properties** from the pop-up menu for detailed information about that call stack entry.

**RELATED TASKS**
Using the Stacks pane

**Controlling program execution**

**Running a program**

You can have a program run until one of the following occurs:

- end of program is reached
- an active breakpoint is hit
- a specific line number is reached
- an exception occurs.

If you run a program, the program will run to completion or until an active breakpoint is hit or until an unmonitored exception occurs.

To run a program, do one of the following:

- Click the run button ( ).
- Select **Debug > Run** from the menu bar.
- Press F5.

When a program runs to completion, the manner in which the debugger terminates changes depending on whether you loaded the program into the debugger without specifying a job or attached the debugger to an existing AS/400 job. If the program was loaded into the debugger without specifying a job, running it to completion will cause termination of the debugger. If the debugger was attached to a job, the debugger must be terminated manually, unless the job under debug finishes. For information about terminating the debugger, see the related topic below.

If you attached the debugger to an existing AS/400 job and specified a program name with the Load Program dialog box **Step Into** control deselected, you will be asked to start your AS/400 application when you run the program.

**RELATED CONCEPTS**
Breakpoints
**RELATED TASKS**
Attaching to an existing AS/400 job from the debugger user interface
Stepping through a program
Running to a location
Terminating a debug session

**Running to a location**
If you run to a location in a program, the program will run to the statement selected unless an active breakpoint is hit, an exception occurs or the end of the program is reached. The program will run to the statement before executing it or any of its procedure calls.

Only executable lines (those that are colored blue) can be run to with Run To Location.

To run a program to a specific line number:
1. Make sure the line to run to is visible in the Source pane by using the scroll bar or cursor keys to locate the line.
2. Run the program to the line by doing one of the following:
   - Right-click on the line to invoke the pop-up menu, then select Run To Location.
   - Click on the line to select it, then select Debug > Run To Location from the menu bar.
   - Click on the line to select it, then select Selected > Run To Location from the menu bar.

**RELATED CONCEPTS**
Breakpoints

**RELATED TASKS**
Stepping through a program
Running a program

**Stepping through a program**
You can use step commands to step through your program a single statement at a time. For an explanation of the step commands, see the related topic below.

To execute a Step Over command, do one of the following:
- Click the step over button ( ) on the toolbar.
- Select Debug > Step Over from the menu bar.
- Press F10.

To execute a Step Debug command, do one of the following:
- Click the step debug button ( ) on the toolbar.
- Select Debug > Step Debug from the menu bar
- Press F7.

**JAVA**
When debugging your Java program, stepping behavior may be irregular when stepping into constructors, or when stepping into or over SystemLoad library functions.
Restarting a program

To start debugging your program from the beginning if your program is stopped, click on \( \text{Restart} \) in the toolbar, select **Debug > Restart** from the menu bar, or press **Ctrl+Shift+F5**.

Depending on the debug session start-up mode or language under debug mode, restarting a program will cause the following:

- If you attached the debugger to a job when you started the debug session, a message will prompt you to call the program again in your AS/400 job before the debug session restarts.
- If you attached the debugger to a job without specifying a program name and then ran the program to termination, you can restart the debugger for additional debug sessions.
- If you loaded the program into the debugger without specifying a job when you started the debug session, the program will restart automatically when you click **Restart**.
- For non-threaded ILE or OPM programs, a message prompts you to call the program in your AS/400 job before the debug session restarts.

**Note:** If you restart an interactive program that produces 5250 screen output, the debugger will run the program to termination before restarting it. You will then have less time than is necessary to dismiss the 5250 screen before the program restarts, and dismissing the 5250 screen will cause termination of the restarted program. When debugging such a program, running to the last statement in the program before restarting will allow for better restart behavior.

Terminating a debug session

To terminate a debug session and exit the debugger, do one of the following:

- Select **File > Exit** from the Distributed Debugger window.
- In the debugger window, use your windowing system’s technique for closing that window. For example, double-click on the upper left corner of the window or press **Alt+F4**. You can also single click the upper right corner.

You may want to end the debugger session for one of the following situations:

- In the Load Program dialog box, you did not specify a program name. If a program name is specified, the debug session will terminate automatically when the program has run to completion. If the program name is not specified, the debug session must be terminated manually.
- If you have passed the point in the program where you intended to debug.
- In the Load Program dialog box, you specified the name of a job that is an active AS/400 job, but it is not the job that you wanted to use.
• You specified the name of a program in the Load Program dialog box, but the program fails when you call it (for example, an unmonitored message occurs when you call the program).

• The debugger is attached to an existing AS/400 job, but the program you wanted to debug has terminated.

**Related Tasks**
Starting the debugger
Loading a program without specifying a job from a command line
Loading a program without specifying a job from the debugger user interface
Attaching to an existing AS/400 job from a command line
Attaching to an existing AS/400 job from the debugger user interface
Restarting a program
Running a program

**Working with threads**

**Debugging multithreaded applications**
To debug secondary threads effectively, in the Source pane bring to focus the source of the code that executes in the secondary threads and set a line breakpoint at the first debuggable statement. Running the program will now stop at the beginning of the thread, which will allow you to set breakpoints for this thread and perform all other debugging functions on it.

When stepping through the thread creation code, additional threads may show in the Stacks pane. To guarantee a view of these threads in the Stacks pane, set a breakpoint in the code which runs in the thread that you want to view. If necessary, add the program which contains the thread you want to view to the Programs pane.

To display the current line of execution in a displayed thread, select **Source > Scroll To Thread Execution Point** from the menu bar. To display the current line of execution of the thread that has caused the debugger to stop (for example, the thread where a breakpoint was hit), select **Source > Show Stopping Thread Location** from the menu bar.

**Related Concepts**
Secondary threads in the debugger

**Related Tasks**
Starting the debugger for a threaded application
Using the Stacks pane
Stepping through a program
Adding a program to the Programs pane

**Related References**
Step commands

**Using the Stacks pane**
The Stacks pane contains a list of call stacks for every thread of execution. You can expand each Call Stack for individual threads to get a complete list of all call stack entries. Once you highlight a particular call stack entry (regardless of its thread of execution), its source is displayed in the Source pane.

To view the source for an entry in the Stacks pane:
1. Click on the Stacks tab to bring the Stacks pane to the foreground.
2. In the Stacks pane, expand the thread that contains the entry that you want to view. Alternatively, you can select Stacks > Expand All so that all entries are visible in the Stacks pane, however, stepping through a program with all stacks expanded will impact performance.
3. If necessary, scroll the Stacks pane to ensure that the entry you want to view is visible. Scroll by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the line.
4. Left-click on the entry you want to view.
5. The source for the selected entry in the Stacks pane is displayed in the Source pane. If this entry is the current entry, the current line is highlighted. Otherwise, the line that called the next innermost entry on the call stack is highlighted.

To display the current line of execution in a displayed thread, select Source > Scroll To Thread Execution Point from the menu bar. To display the current line of execution of the thread that has caused the debugger to stop (for example, the thread where a breakpoint was hit), select Source > Show Stopping Thread Location from the menu bar. The Scroll To Thread Execution Point and Show Stopping Thread Location actions apply when you are debugging the only thread in a single-threaded program or a thread in a multithreaded program. For single-threaded programs, both actions accomplish the same task.

**RELATED CONCEPTS**
Distributed Debugger: Panes

**RELATED TASKS**
Debugging multithreaded applications
Viewing the contents of the Call Stack

**Viewing the contents of the Call Stack**
You can view information for the active programs, modules, procedures, or functions on a thread’s call stack.

To view stack information, click on the Stacks pane to bring it in focus and expand the appropriate thread entry. Right-click on the call stack entry that you want to view and select Properties from the pop-up menu for detailed information about that call stack entry.

**RELATED TASKS**
Using the Stacks pane

**Inspecting variables**

**Adding a variable or expression to the Monitors pane**
From the Source pane or the Monitors menu, you can add variables and expressions to the Monitors pane, so that you can keep track of how their contents change during program execution. You can add multiple variables and expressions to the Monitors pane from the Monitors menu.

Each thread has local variables which can also be monitored in the Locals pane. By default, all local variables in scope are added to the Locals pane. When a thread is terminated, its local variables no longer appear in the Locals pane.

To add a variable or expression to the Monitors pane from the Source pane:
1. In the Source pane, select or double-click the variable or expression you want to monitor.
2. Right-click on the highlighted variable, and select Add to Program Monitor from the pop-up menu.

To add a variable or expression to the Monitors pane from the Monitors menu:
1. In the Source pane, selecting or double-clicking the variable or expression you want to monitor is optional. If this step is completed, the variable or expression will appear in the Enter expression to be evaluated field of the Monitor Expression dialog box and making an entry in this field will not be required.
2. Select Monitors > Monitor Expression from the menu bar to invoke the Monitor Expression dialog box.
3. If you did not complete step 1, enter the variable or expression you want to monitor in the Enter expression to be evaluated field.
4. The Program monitor radio button will default to and remain selected.
5. Click OK to add the variable or expression to the monitor and dismiss the dialog box.

To add multiple variables or expressions to the Monitors pane from the Monitors menu:
1. Select Monitors > Monitor Expression from the menu bar.
2. In the dialog box, enter the variable or expression you want to monitor.
3. Click Monitor to add the variable to the monitor.
4. Repeat steps 2 and 3 until you have added all the variables or expressions you want to monitor.
5. Click Cancel to dismiss the dialog box.

Note: If you wish to monitor a specific array or element, you can use the Monitor Expression dialog from the Monitors menu.

RELATED CONCEPTS
Distributed Debugger: Panes
Distributed Debugger: Monitors

RELATED TASKS
Viewing the contents of a variable or expression
Changing the contents of a variable or expression
Enabling and disabling a monitored variable
Changing the representation of monitor contents
Deleting a monitored variable
Dereferencing pointers

Viewing the contents of a variable or expression
You can view the contents of a variable or expression in the Locals pane or the Monitors pane, if you have added the variable there. By default, all local variables in scope are added to the Locals pane.

To view the contents of a variable or expression in the Locals pane:
1. Click on the Locals tab to bring the Locals pane into focus.
2. Expand the thread in the Locals pane where the local variable you want to view appears. Alternatively, select Locals > Expand All so that all variables are visible in the Locals pane.
3. If necessary, scroll the Locals pane to ensure that the variable you want to view is visible in the pane. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the variable.

4. If your variable is a class, structure, or array, it can be expanded to show its individual elements.

5. If desired, change the representation of the variable: right-click on the variable and select **Representation** from the pop-up menu and then select a representation from the Monitor Representation dialog box.

To view the contents of a variable or expression you have already added to the Monitors pane:

1. Click on the **Monitors** tab to bring the Monitors pane into focus.

2. If necessary, scroll the Monitors pane to ensure that the variable you want to view is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the variable.

3. If your variable is a class, structure, or array, it can be expanded to show its individual elements.

4. If desired, change the representation of the variable: right-click on the variable and select **Representation** from the pop-up menu and then select a representation from the Monitor Representation dialog box.

If a variable or expression is not in scope, a message displays in the Monitors pane instead of a value.

***Note:** For ILE RPG and ILE COBOL, local variables cannot be displayed in the Locals pane. However, they can be displayed in the Monitors pane.

***Important:** After exiting a program block, variables out of scope may still be shown in the Monitors pane.

You can also view the contents of variables in the Source pane with Tool Tip Evaluation. To enable Tool Tip Evaluation, see the related topic below.

**Related Tasks**
- Adding a variable or expression to the Monitors pane
- Enabling and using Tool Tip Evaluation for variables
- Expanding and contracting lists

**Changing the contents of a variable or expression**

To change the contents of a variable in the Locals pane or Monitors pane:

1. Click on the pane’s tab to bring the pane to the foreground.

2. In the pane, expand the thread where the variable you want to change appears. Alternatively, select **Monitors > Expand All** or **Locals > Expand All**, depending on the pane that you are in, so that all variables are visible in the pane.

3. If the variable that you want to change is a class, structure, or array, expand it to show its individual elements.

4. If necessary, scroll the pane to ensure that the variable you want to change is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the variable.

5. Do one of the following:
   - Double-click on the variable or variable element.
   - Right-click on the variable and select **Edit** from the pop-up menu.
6. Enter a value that is valid for the current representation of that variable or variable element. The debugger checks for a valid value.

7. Press Enter to submit the change.

If the same variable is displayed in more than one monitor, the changes you make are reflected in all monitors. However, if two variables with the same name qualify to different programs or threads, they are considered to be two different variables. Therefore, a change to variable a in program 1 or thread 1 does not affect the value of variable a in program 2 or thread 2.

JAVA You cannot change the contents of String objects in Java.

RELATED TASKS
Adding a variable or expression to the Monitors pane
Viewing the contents of a variable or expression

Enabling and disabling a monitored variable
You can enable or disable the monitoring of a variable from the Monitors pane, Locals pane, or Selected menu. The advantage of disabling a monitored variable instead of deleting it is that it is easier to enable a monitored variable than to recreate it.

Enabled variables are indicated with a red diamond (🔴). Disabled variables are indicated with a gray diamond (_GRAY_).

To enable or disable a monitored variable from the Monitors or Locals pane, do the following:
1. Click on the pane’s tab to bring the pane to the foreground.
2. In the pane, expand the thread where the variable you want to enable or disable appears. Alternatively, select Monitor > Expand All or Locals > Expand All, depending on the pane that you are in, so that all variables are visible in the pane.
3. If necessary, scroll the pane to ensure that the variable you want to enable or disable is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the variable.
4. Right-click on the variable you want to enable or disable.
5. Select Enable or Disable from the pop-up menu.

To enable or disable a monitored variable from the Selected menu, do the following:
1. In the Monitors pane or Locals pane, select the monitor you want to enable or disable.
2. Select Selected > Enable or Selected > Disable from the menu bar.

To enable or disable multiple monitored variables, do the following:
1. In the Monitors pane or Locals pane, select the monitors you want to enable or disable. To select more than one monitor at a time, press the Ctrl key on the keyboard while making all selections.
2. Do one of the following:
   • Select Selected > Enable or Selected > Disable from the menu bar.
   • Press the Shift key and right-click on the selected variables and select Enable or Disable from the pop-up menu.
Monitors

Adding a variable or expression to the Monitors pane
Viewing the contents of a variable or expression
Changing the contents of a variable or expression
Changing the representation of monitor contents

**Changing the representation of monitor contents**
You can change the representation of existing entries in the Monitors pane or Locals pane. You can change the default representation for future entries in the Application Preferences window.

To change the representation of a variable or expression in the Monitors pane or Locals pane:
1. Click on the pane’s tab to bring the pane to the foreground.
2. Right-click on the variable or expression you want to change the representation of.
3. Select *Representation* from the pop-up menu. The Monitor Representation dialog box appears.
4. Select the representation you want from the list of available representations.
5. Click *OK* to change the representation and dismiss the Monitor Representation dialog box.

To change the default representation of variables or expressions:
1. Select *File > Preferences* from the main menu bar. The Application Preferences window appears.
2. In the left-hand pane of the Application Preferences window, go to *Debug > program > Default Monitor Representation*, where *program* is the name of a program loaded in the Distributed Debugger you want to change the default representation for.
3. Change the representations for variable types by clicking on the representation associated with a variable type and selecting a representation from the list.
4. If you want these representations to become the default for the Distributed Debugger to use when no program profile is available, click *Debugger Defaults*.
5. Click *OK* to change the default representations and dismiss the Application Preferences window.

The default representations of variables and expressions in programs you have previously debugged will not be affected by these changes.

**Related Tasks**
Adding a variable or expression to the Monitors pane

**Deleting a monitored variable**
You can delete a monitored variable from the Monitors pane, Locals pane, or Selected menu.

To delete a monitored variable from the Monitors or Locals pane, do the following:
1. Click on the pane’s tab to bring the pane to the foreground.
2. In the pane, expand the thread where the variable you want to enable or disable appears. Alternatively, select Monitors > Expand All or Locals > Expand All, depending on the pane that you are in, so that all variables are visible in the pane.

3. If necessary, scroll the pane to ensure that the variable you want to delete is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the variable.

4. Right-click on the variable you want to delete.

5. Select Delete from the pop-up menu.

To delete a monitored variable from the Selected menu, do the following:
1. In the Monitors pane, select the variable you want to delete.
2. Select Selected > Delete from the menu bar.

To delete multiple monitored variables, do the following:
1. In the Monitors pane, select the variables you want to delete. To select more than one monitor at a time, press the Ctrl key on the keyboard while making all selections.
2. Do one of the following:
   - Select Selected > Delete from the menu bar.
   - Press the Shift key and right-click on the selected variables and select Delete from the pop-up menu.

RELATED CONCEPTS
Monitors

RELATED TASKS
Adding a variable or expression to the Monitors pane
Viewing the contents of a variable or expression
Changing the contents of a variable or expression
Enabling and disabling a monitored variable
Changing the representation of monitor contents

Dereferencing pointers
You can dereference pointers in the Locals pane or pointers that have been added to the Monitors pane.

Pointers are indicated with a green arrow (▷).

To dereference a pointer from the Monitors or Locals pane, do the following:
1. Click on the pane’s tab to bring the pane to the foreground.
2. In the pane, expand the thread where the pointer you want to dereference appears. Alternatively, select Monitors > Expand All or Locals > Expand All, depending on the pane that you are in, so that all entries are visible in the pane.
3. If necessary, scroll the pane to ensure that the pointer you want to dereference is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the pointer.
4. Right-click on the pointer you want to dereference.
5. Select De-reference from the pop-up menu.

To dereference a pointer from the Selected menu, do the following:
1. In the Monitors pane or Locals pane, select the pointer you want to dereference.
2. Select **Selected > De-reference** from the menu bar.

To dereference multiple pointers, do the following:

1. In the Monitors pane or Locals pane, select the pointers you want to dereference. To select more than one pointer at a time, press the Ctrl key on the keyboard while making all selections.

2. Do one of the following:
   - Select **Selected > De-reference** from the menu bar.
   - Press the Shift key and right-click on the selected pointers and select **De-reference** from the pop-up menu.

**RELATED CONCEPTS**

Monitors

**RELATED TASKS**

Adding a variable or expression to the Monitors pane
Viewing the contents of a variable or expression
Changing the contents of a variable or expression
Changing the representation of monitor contents

### Working with the debugger user interface

**Enabling and using Tool Tip Evaluation for variables**

Tool Tip Evaluation for variables provides you with a quick way to view the contents of variables in the Source pane. Tool Tip Evaluation also displays expression context, or the thread number that a variable is qualified to. When you point at a variable in the Source pane, a pop-up appears displaying the contents of that variable. This feature is enabled by default when you first start the debugger.

To enable or disable Tool Tip Evaluation, select or deselect **Source > Allow Tool Tip Evaluation** from the menu bar. A check mark will appear next to the Allow Tool Tip Evaluation menu item to indicate it is enabled. When the feature is disabled, the menu item has no check mark next to it.

To enable or disable Tool Tip Evaluation as the default:

1. Select **File > Preferences** from the main menu to invoke the Application Preferences window.
2. Select **Debug** from the list of preferences to set.
3. Select or deselect **Allow Tool Tip Evaluation** from the **Debugger Defaults** section.
4. Click **OK** to enable or disable the tool tip monitor and dismiss the window.

**RELATED CONCEPTS**

Distributed Debugger: Monitors
Distributed Debugger: Panes

**RELATED TASKS**

Viewing the contents of a variable or expression

**RELATED REFERENCES**

Expressions qualified to thread number
Expanding and contracting lists

Much of the information presented in the Distributed Debugger is in the form of expandable lists. These lists appear in panes such as the Stacks pane and the Breakpoints pane. They also appear in dialog boxes such as the Debugger Settings dialog box. In most cases, you can expand and contract these lists one level at a time or expand or contract all levels of the list at once.

These lists have nodes that will expand or contract one level of the list when you click on them. The appearance of the nodes depends on the Platform setting you selected in the Appearances part of the Application Preferences window.

If your selected platform is Windows:
- Expand one level of a list by clicking on the plus icon (➕).
- Contract one level of a list by clicking on the minus icon (➖).

If your selected platform is CDE/Motif:
- Expand one level of a list by clicking on the plus icon.
- Contract one level of a list by clicking on the minus icon.

If your selected platform is Metal:
- Expand one level of a list by clicking on the white circle.
- Contract one level of a list by clicking on the dark circle.

To expand all levels of a list in a pane, select Expand All from the menu for that pane. For example, to see all set breakpoints, select Breakpoints > Expand All.

To collapse all levels of a list in a pane, select Collapse All from the menu for that pane. For example, to hide the list of all set breakpoints, select Breakpoints > Collapse All.

Breakpoints

Breakpoints are temporary markers you place in your program to tell the debugger to stop your program whenever execution reaches that point. For example, if a particular statement in your program is causing problems, you could set a breakpoint on the line containing the statement, then run your program. Execution stops at the breakpoint before the statement is executed. You can then check the contents of variables and the call stack, and step over (execute) the statement to see how the problem arises.

The Distributed Debugger supports the following types of breakpoints:
- **Line breakpoints** are triggered before the code at a particular line in a program is executed.
- **Watch breakpoints** compare the content of a storage location (for example, the content of a program variable) after the execution of each statement. They triggered when the contents of the storage location of a variable changes.

You can set conditions on line and watch breakpoints. For line breakpoints when you run the program, execution stops at the breakpoint before the statement is executed if the breakpoint condition is met. For watch breakpoints when you run the program, execution stops at the breakpoint after the statement is executed if the breakpoint condition is met.
**Line breakpoints**

By setting line breakpoints, you can stop your program at the line number you select. You can only set breakpoints on lines containing executable statements. If there is more than one executable statement on a line, the program will only stop once before executing the first statement.

In CL, setting breakpoints on consecutive lines will not always stop the program multiple times, because some statements are mapped to the same line number, such as statements that span multiple lines or DO ENDDO combinations.

**Watch breakpoints**

Use this type of breakpoint when you want to see where and how a variable is being changed in your program. A watch breakpoint can be deleted, but not modified.

By setting watch breakpoints, you can stop your program when the values of variables change. The watch is triggered when the content of the storage location of a variable changes.

When the content of the watched storage location changes, the program stops at the first executable statement following the change, and this line is highlighted in the Source pane. If the program that caused this variable to be changed has not been added to the debugger, it is automatically added if the program contains debug data, and you have sufficient authorization to the program.

Watch breakpoints are always set job-wide but qualified to a specific thread.

**Notes:**

- You may encounter a watch breakpoint within a program that you do not have sufficient authority to debug or that does not contain debug data. In this case, you would receive a message that specifies the program and location where the watch condition was encountered. Program execution resumes when you click the message’s OK button.
- Watching variables may slow your program, because the value of the variable must be checked on each statement.

**Watch Breakpoints in multithreaded applications**

When debugging a multithreaded application, you need to consider the following:

- Watch breakpoints are set based on the currently stopped position within a selected or specified thread. In this case, the debugger uses the selected or specified thread to qualify the watched variable.
Watch breakpoints may be hit in any thread, no matter what thread they are set for.

Watch breakpoints remain set even if the thread in which they were originally set terminates.

**Related Tasks**
- Setting a watch breakpoint

**Related References**
- Characteristics of watches

## Characteristics of watches

Consider the following characteristics of watches:

**Watch breakpoints cannot be modified**

A watch breakpoint can be deleted, but not modified.

**Maximum number of watches**

Watches are set on a system-wide basis, with a maximum number of 256 watches that can be active simultaneously. This number includes watches set by the system. If a variable crosses a page boundary, two watches are used internally to watch the storage locations. The maximum number of variables that can be watched simultaneously on a system-wide basis ranges from 128 to 256.

Depending on overall system use, you may be limited in the number of watch conditions you can set at a given time. If you try to set a watch condition while the maximum number of active watches across the system is exceeded, you will receive an error message and the watch condition is not set.

**Scope of watched variables**

Watch conditions can only be set when a program is stopped under debug mode, and the variable to be watched is in scope. If this is not the case, an error message is issued when a watch is requested.

Once the watch condition is set, the address of a storage location watched does not change. If a watch is set on a temporary location, such as the automatic storage of an ILE C or C++ procedure, which can be reused after the procedure ends, it may result in invalid watch-condition notifications.

A watch condition may be registered although the watched variable is no longer in scope. You must not assume that a variable is in scope just because a watch condition is reported. **Overlapping watch locations** Two watch locations in the same job must not overlay in any way. Two watch locations in different jobs must not start at the same storage address; otherwise, overlap is allowed. If these restrictions are violated, an error message is issued. **Watch breakpoints** After the command is successfully run, your application is stopped if a program in your session changes the contents of the watched storage location.

If the program has debug data, and a Source view is available, it is shown. The source line of the statement that is about to be run when the content change at the storage location is detected is highlighted. A message indicates which watch condition is satisfied.
If the program cannot be debugged, the text area of the display remains blank. **Programs added to a debug session** Eligible programs are automatically added to the debug session if they cause the watch stop.

**RELATED CONCEPTS**
Watch breakpoints

**RELATED TASKS**
Setting a watch breakpoint

**Thread-specific and job-wide line breakpoints**
You can set line breakpoints for a specific thread or job-wide for all threads. You cannot set a thread-specific line breakpoint and a job-wide line breakpoint at the same statement. This limitation may lead to unexpected behavior when you try to set, enable, or modify a breakpoint for which another breakpoint of the opposite type is already active.

The following rules apply when setting or enabling line breakpoints:
- If you try to set or enable a breakpoint for a line for which another breakpoint of the opposite type is enabled, the job-wide breakpoint will take precedence over the thread-specific breakpoint.
- If you try to set or enable a breakpoint for a line for which another breakpoint of the opposite type is disabled, the new breakpoint will be set, and the breakpoint of the opposite type will remain disabled.

The following rules apply when modifying line breakpoints to the opposite type (for example, changing a thread-specific breakpoint to a job-wide breakpoint):
- If you try to modify a breakpoint to the opposite type, for a line for which another breakpoint of the opposite type is enabled, the job-wide breakpoint will take precedence over the thread-specific breakpoint.
- If you try to modify a breakpoint to the opposite type, for a line for which another breakpoint is disabled, the new breakpoint will be set. The disabled breakpoint will remain disabled if it is of the opposite type. It will be deleted if it is of the same type.

**RELATED CONCEPTS**
Breakpoints

**RELATED TASKS**
Enabling and disabling breakpoints
Modifying line breakpoint properties
Setting a line breakpoint

**Setting Breakpoints**

**Setting a line breakpoint**
You can set line breakpoints from the Source pane, the Source menu, the Selected menu, and the Breakpoints menu.

To set a line breakpoint in the Source pane:
1. Make sure the appropriate line is visible in the Source pane by using the scrollbar, the PageUp and PageDown keys, or cursor keys to locate the line.
2. Do one of the following:
   - Double-click on the line number in the prefix area of the line.
• Right-click on the line you want to set a breakpoint on, and select **Set Breakpoint** from the pop-up menu.

• Select the line you want to set a breakpoint on, and press F9.

To set a line breakpoint from the Source menu, select the line you want to set a breakpoint on, and select **Source > Set Breakpoint** from the menu bar, or:

1. Select **Source > Set Line Breakpoint** from the menu bar to invoke the Line Breakpoint dialog box.

2. Enter the name of the module in which you want to set a breakpoint in the **Program** entry field of the dialog box. If this module is loaded, you can select it from the pull-down list in the **Program** entry field.

3. Choose or enter the object, class or source file that is associated with the module or routine specified in the **Program** entry field and contains the line where the breakpoint is to be set from the **Source** pull-down list.

4. Choose the source file containing the code for the object or class file from the **Source (optional)** pull-down list. (This step is optional.)

5. Enter the line number within the source file where you want to place a breakpoint in the **Line** entry field.

6. Set any optional parameters that you want for the breakpoint.

7. Click **OK** to set the breakpoint and dismiss the Line Breakpoint dialog box. Alternatively, use the **Set** button to set the breakpoint without dismissing the Line Breakpoint dialog box.

To set a line breakpoint from the Selected menu:

1. Make sure the appropriate line is visible in the Source pane by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the line.

2. Select the line you want to set a breakpoint on, and select **Selected > Set Breakpoint** from the menu bar.

To set a line breakpoint from the Breakpoints menu:

1. Click on the Breakpoints pane to bring the Breakpoints menu in focus.

2. Select **Breakpoints > Set Line** from the menu bar.

3. Enter the name of the module or routine in which you want to set a breakpoint in the **Program** entry field in the Line Breakpoint dialog box. If this module or routine is loaded, you can select it from the pull-down list in the **Program** entry field.

4. Choose or enter the object, class or source file that is associated with the module or routine specified in the **Program** entry field and contains the line where the breakpoint is to be set from the **Source** pull-down list.

5. Choose the source file containing the code for the object or class file from the **Source (optional)** pull-down list. (This step is optional.)

6. Enter the line number within the source file where you want to place a breakpoint in the **Line** entry field.

7. Set any optional parameters that you want for the breakpoint.

8. Click **OK** to set the breakpoint and dismiss the Line Breakpoint dialog box. Alternatively, use the **Set** button to set the breakpoint without dismissing the Line Breakpoint dialog box.

**RELATED CONCEPTS**
Breakpoints

**RELATED TASKS**
Setting multiple breakpoints
Setting a conditional breakpoint
Modifying line breakpoint properties
Enabling and disabling breakpoints
Deleting a breakpoint
Viewing set breakpoints

**Setting a watch breakpoint**

Note: Watch breakpoints are not supported for Java classes.

You can set watch breakpoints from the Breakpoints menu by doing the following:

1. Select **Breakpoints > Set Watch** from the menu bar to invoke the Watch Breakpoint dialog box.
2. Enter the variable you want to watch in the **Expression** entry field of the dialog box, and enter a value in the **Bytes to Monitor** entry field. To watch a variable in its defined length, enter 0 in the **Bytes to Monitor** entry field.
3. Set any optional parameters that you want for the watch.
4. Click **OK** to set the watch and dismiss the Watch Breakpoint dialog box.
   Alternatively, use the **Set** button to set the watch without dismissing the Watch Breakpoint dialog box.

**RELATED CONCEPTS**
Breakpoints

**RELATED REFERENCES**
Characteristics of watches

**Setting a conditional breakpoint**

When you set a breakpoint, you can specify the parameters or conditions for that breakpoint.

To set a conditional breakpoint:

1. Select the type of breakpoints you want to set from the **Breakpoints** menu or, for setting multiple line breakpoints only, from the **Source** menu.
2. From the resulting breakpoint dialog box, complete any or all optional parameters (in the **Optional Parameters** section) that you want as conditions for your breakpoint.
3. Click **OK** to set the conditional breakpoint and dismiss the dialog box.

**RELATED CONCEPTS**
Breakpoints

**RELATED TASKS**
Setting multiple breakpoints
Modifying line breakpoint properties
Enabling and disabling breakpoints
Deleting a breakpoint
Viewing set breakpoints

**Setting multiple breakpoints**

You can set several breakpoints with the same optional parameters from either of the breakpoint dialog boxes.

To set multiple occurrences of either type of breakpoint from the same dialog box:
1. Select the type of breakpoints you want to set from the Breakpoints menu or, for setting multiple line breakpoints only, from the Source menu.

2. From the resulting breakpoint dialog box, enter the required information for the first breakpoint. Change any fields in the Optional Parameters section of the dialog box, as desired.

3. Click on Set. The settings are saved for the current breakpoint.

4. For each additional breakpoint, change the information for the new breakpoint (for example, new line number or expression) and click on Set.

5. After you have set the last breakpoint, click on Cancel to dismiss the dialog box.

Note: You cannot set multiple breakpoints on the same line.

**RELATED CONCEPTS**

Breakpoints

**RELATED TASKS**

Modifying line breakpoint properties
Enabling and disabling breakpoints
Setting a conditional breakpoint
Deleting a breakpoint
Viewing set breakpoints

### Viewing set breakpoints

A list of breakpoints you have set is kept in the Breakpoints pane for the job you are debugging. This list is originally collapsed and can be expanded to show your set breakpoints. The expanded list of breakpoints is divided into the types of breakpoints you may have set. Expanding each type of breakpoint will provide you with a list of breakpoints for that type.

To view the list of breakpoints:

1. In the Breakpoints pane, expand the list of breakpoints until you see the breakpoint you want to view. Alternatively, select Breakpoints > Expand All so that all breakpoints are visible in the Breakpoints pane.

2. If necessary, scroll the Breakpoints pane to ensure that the breakpoint you want to view is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the breakpoint.

To view the properties of a breakpoint, right-click on the desired breakpoint and select Breakpoint Properties from the pop-up menu.

Note: A set breakpoint (enabled or disabled) may appear on a different line number in the Source pane, depending on which view you are in. If a breakpoint is set on a line, its location in the Source pane will change if you change the Source pane view. When viewing the properties of a breakpoint in the Breakpoints pane, the line number will be associated with the view in which the breakpoint was set.

**RELATED CONCEPTS**

Breakpoints

**RELATED TASKS**

Setting multiple breakpoints
Modifying line breakpoint properties
Enabling and disabling breakpoints
Deleting a breakpoint

Modifying line breakpoint properties
You can change the following properties of a line breakpoint:
• Which threads the breakpoint applies to.
• How often the debugger should skip the breakpoint (the frequency).
• Whether to stop at a breakpoint only when a given expression is true.
You can also change the Required parameters fields for a breakpoint. Changing these fields results in the existing breakpoint being deleted and a new breakpoint being set.

To change a line breakpoint’s properties in the Breakpoints pane:
1. Click on the Breakpoints tab to bring the Breakpoints pane to the foreground.
2. In the Breakpoints pane, expand the list of line breakpoints until you see the breakpoint you want to modify. Alternatively, select Breakpoints > Expand All so that all breakpoints are visible in the Breakpoints pane.
3. If necessary, scroll the Breakpoints pane to ensure that the breakpoint you want to modify is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the breakpoint.
4. Right-click on the breakpoint you want to modify.
5. Select Modify Breakpoint from the pop-up menu. A breakpoint dialog box corresponding to the breakpoint type appears displaying the current settings for the breakpoint.
6. Change the breakpoint’s properties in the breakpoint dialog box.
7. Click OK to set the breakpoint with modified properties and dismiss the breakpoint dialog box. Alternatively, use the Set button to set the breakpoint with modified properties and then dismiss the breakpoint dialog box by clicking Cancel.

To change a line breakpoint’s properties from the Selected menu:
1. Click on the Breakpoints tab to bring the Breakpoints pane to the foreground.
2. In the Breakpoints pane, expand the list of line breakpoints until you see the breakpoint you want to modify.
3. Click on the breakpoint you want to modify.
4. Select Selected > Modify Breakpoint from the menu bar. A breakpoint dialog box corresponding to the breakpoint type appears displaying the current settings for the breakpoint.
5. Change the breakpoint’s properties in the breakpoint dialog box.
6. Click OK to set the breakpoint with modified properties and dismiss the breakpoint dialog box. Alternatively, use the Set button to set the breakpoint with modified properties and then dismiss the breakpoint dialog box by clicking Cancel.

Related concepts
Breakpoints

Related tasks
Setting multiple breakpoints
Enabling and disabling breakpoints
Deleting a breakpoint
Viewing set breakpoints

Enabling and disabling breakpoints

You can disable a breakpoint so that it does not stop execution, and then enable it again later. Information about the breakpoint (such as type, location, condition, and frequency) is saved by the Distributed Debugger when the breakpoint is disabled. Since this is not true when the breakpoint is deleted, the advantage of disabling a breakpoint instead of deleting it is that it is easier to enable a breakpoint than to recreate it. Enabled breakpoints are indicated with a red dot ( ). Disabled breakpoints are indicated with a gray dot ( ).

You can enable or disable a single breakpoint from the Breakpoints pane, Source pane, Source menu, or Selected menu. You can also enable or disable all breakpoints from the Breakpoints menu.

To enable or disable a single breakpoint from the Breakpoints pane:
1. Click on the Breakpoints tab to bring the Breakpoints pane to the foreground.
2. In the Breakpoints pane, expand the list of breakpoints until you see the breakpoint you want to enable or disable. Alternatively, select Breakpoints > Expand All so that all breakpoints are visible in the Breakpoints pane.
3. If necessary, scroll the Breakpoints pane to ensure that the breakpoint you want to enable or disable is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the breakpoint.
4. Do one of the following:
   • Right-click on the breakpoint you want to enable or disable and select Enable Breakpoint or Disable Breakpoint from the pop-up menu.
   • Select the breakpoint and press Ctrl+F9 to enable or disable the breakpoint, depending on its current state.

To enable or disable a breakpoint from the Source pane:
1. Scroll to the line which contains the breakpoint you want to enable or disable.
2. Do one of the following:
   • Right-click on the breakpoint you want to enable or disable and select Enable Breakpoint or Disable Breakpoint from the pop-up menu.
   • Select the breakpoint and press Ctrl+F9 to enable or disable the breakpoint, depending on its current state.

To enable or disable a breakpoint from the Source menu:
1. Do one of the following:
   • In the Source pane, scroll to and select the line which contains the breakpoint you want to enable or disable.
   • In the Breakpoints pane, select the breakpoint you want to enable or disable.
2. Select Source > Enable Breakpoint or Source > Disable Breakpoint from the menu bar.

To enable or disable a breakpoint from the Selected menu:
1. Do one of the following:
   • In the Source pane, scroll to and select the line which contains the breakpoint you want to enable or disable.
   • In the Breakpoints pane, select the breakpoint you want to enable or disable.
2. Select **Selected > Enable Breakpoint** or **Selected > Disable Breakpoint** from the menu bar.

To enable all breakpoints, click on the **Breakpoints** tab to enable the Breakpoints pull-down menu and select **Breakpoints > Enable All Breakpoints** from the menu bar.

To disable all breakpoints, click on the **Breakpoints** tab to enable the Breakpoints pull-down menu and select **Breakpoints > Disable All Breakpoints** from the menu bar.

**RELATED CONCEPTS**

Breakpoints

**RELATED TASKS**

Deleting a breakpoint

### Deleting a breakpoint

You can delete single breakpoints from the Breakpoints pane, Source pane, Source menu, or Selected menu. You can also delete all breakpoints from the Breakpoints menu. If you delete a breakpoint, all information on it is lost. If you do not want to lose your breakpoint information, but do not want the breakpoint to stop program execution, disable the breakpoint instead. For information on disabling breakpoints, see the related topic below.

To delete a single breakpoint from the Breakpoints pane:

1. Click on the **Breakpoints** tab to bring the Breakpoints pane to the foreground.
2. In the Breakpoints pane, expand the list of breakpoints until you see the breakpoint you want to delete. Alternatively, select **Breakpoints > Expand All** so that all breakpoints are visible in the Breakpoints pane.
3. If necessary, scroll the Breakpoints pane to ensure that the breakpoint you want to delete is visible. Do this by using the scroll bar, the PageUp and PageDown keys, or cursor keys to locate the breakpoint.
4. Do one of the following:
   * Right-click on the breakpoint you want to delete and select **Delete Breakpoint** from the pop-up menu.
   * Select the breakpoint and press **F9**.

To delete a breakpoint from the Source pane:

1. Scroll to the line which contains the breakpoint you want to delete.
2. Do one of the following:
   * Right-click on the breakpoint you want to delete and select **Delete Breakpoint** from the pop-up menu.
   * Select the breakpoint and press **F9**.
   * For line breakpoints only, double-click on the line number in the prefix area of the line containing the breakpoint.

To delete a breakpoint from the Source menu:

1. Do one of the following:
   * In the Source pane, scroll to and select the line which contains the breakpoint you want to delete.
   * In the Breakpoints pane, select the breakpoint you want to delete.
2. Select **Source > Delete Breakpoint** from the menu bar.
To delete a breakpoint from the Selected menu:
1. Do one of the following:
   - In the Source pane, scroll to and select the line which contains the breakpoint you want to delete.
   - In the Breakpoints pane, select the breakpoint you want to delete.
2. Select Selected > Delete Breakpoint from the menu bar.

To delete all breakpoints, click on the Breakpoints tab to enable the Breakpoints pull-down menu and select Breakpoints > Delete All Breakpoints from the menu bar.

**RELATED CONCEPTS**
Breakpoints

**RELATED TASKS**
Enabling and disabling breakpoints

---

**Debugging servlets, EJB, and JSP**

**Debugging servlets, JSP, and EJB**

Debugging web artifacts, such as servlets, Enterprise Java Beans (EJB), or JavaServer Pages (JSP), requires that the following be installed and used during debug preparation.

**On the client machine:**
- IBM Distributed Debugger for AS/400
- Object Level Trace
- WebSphere Application Server, Version 3.02 or 3.5 Administrator’s Console

**On the AS/400 server:**
- WebSphere Application Server, Version 3.02 or 3.5

For more information about setting up web artifact debugging, please see the related topics below.

**RELATED TASKS**
Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP
Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP
Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging
Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

**RELATED REFERENCES**
Preparing servlets, EJB, and JSP for debugging
- An EJB example
- A servlet example
- A JavaServer Page example

---

54 AS/400 Debugging
Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP

For debugging web artifacts, such as servlets, Enterprise Java Beans (EJB), or JavaServer Pages (JSP), that are being serviced by WebSphere Application Server V3.02, set up the application server by editing the file dertr11j.properties, which can be found in the following IFS location on your AS/400 server:

/qibm/userdata/webasadv/myuserid/properties

where myuserid is ‘default’, unless you are using an additional administrative server. For more information about administrative server creation, please see the WebSphere Application Server for AS/400 V3.02 documentation.

In dertr11j.properties, the following values must be updated to set up Object Level Trace (OLT):

1. Edit data.ctrlhost to specify the IP address or machine name of the workstation on which your OLT client user interface is running.
2. Edit data.ctrlport to specify the port number on your workstation on which your OLT client user interface is running. This step is not required if the port number in data.ctrlport matches the default OLT port number (2102).
3. Edit data.dbghost to specify the IP address or machine name of the workstation on which your OLT client user interface is running, exactly as the entry was specified for data.ctrlhost.
4. Ensure that the data.dbgport port number is set to 8001.
5. Edit data.trchost to specify the IP address or machine name of the workstation on which your OLT client user interface is running, exactly as the entry was specified for both data.ctrlhost and data.dbghost.
6. Edit data.trcport to specify the port number on your workstation on which your OLT client user interface is running. This step is not required if the port number in data.trcport matches the default OLT port number (2102).
7. Ensure that the data.dbgoption is set to 3.

RELATED CONCEPTS
Debugging servlets, JSP, and EJB

RELATED TASKS
Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP
Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging
Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

RELATED REFERENCES
Preparing servlets, EJB, and JSP for debugging
An EJB example
A servlet example
A JavaServer Page example
Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP

For debugging web artifacts, such as servlets, Enterprise Java Beans (EJB), or JavaServer Pages (JSP), that are being serviced by WebSphere Application Server V3.5, ensure that the application server has been installed.

**RELATED CONCEPTS**
Debugging servlets, JSP, and EJB

**RELATED TASKS**
Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP
Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging
Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

**RELATED REFERENCES**
Preparing servlets, EJB, and JSP for debugging
An EJB example
A servlet example
A JavaServer Page example

Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging

When preparing to debug a web artifact, such as a servlet, Enterprise Java Bean (EJB), or JavaServer Page (JSP), begin by compiling the object with the appropriate Java compiler options. For more information on preparing these web artifacts, please see the related topics below.

Once the web artifact is compiled with the appropriate options, you must put the web artifact’s location in the application server CLASSPATH, start the Object Level Trace (OLT) viewer, ensure that Step-by-step Debugging Mode is set, then start the application server, and invoke the web artifact. After you complete these steps, you will be prompted to step into the appropriate method. After selecting this method, clicking OK will launch the debugger which will open at the first executable line in the method.

**Note:** To avoid holding up other AS/400 jobs with the job that is created for debugging, it is recommended that you create a separate application server instance for the debug session and deploy your servlet, JSP, or Enterprise Java Bean into it. This way, you do not impact the execution of other servlets, JSP files, or Enterprise Java Beans. This is accomplished by using a different port number for the debugging session.

To set the CLASSPATH and SOURCEPATH in the WebSphere Application Server for AS/400, do the following:
1. Start the WebSphere Administrative Console through one of the following methods:
   - Select Programs > IBM WebSphere > Application Server V.xx > Administrator’s Console, where xx is the application server version number, from the Windows Start menu.
• From a command line, issue the `adminclient <host> <port>` from the `<install-root>` directory, where `<install-root>` is the directory where the WebSphere Administrative Console is installed, `<host>` is the host name of your AS/400 system, and `<port>` is the AS/400 port number that you intend to use.

2. From the administrative console, click the **Topology** tab to view the Topology tree.

3. Use the tree to locate and select the application server that will run the web artifact. The settings for the application server appear on the right-hand side of the console.

4. Specify the directory or directories in which the debuggable class files are located, using the `-classpath` Java Virtual Machine property for the application server which will host the debuggable Enterprise Java Beans, servlets or JSP files. On the **General** tabbed page of the application server’s properties, add the `-classpath` option to the **Command line arguments** field:

   ```
   -classpath /qibm/userdata/webasadv/myuserid/mydirectory
   ```

   where `/qibm/userdata/webasadv/myuserid/mydirectory` is the root directory containing the debuggable class files.

   **Note:** In most cases, `myuserid` will be default. For example, if you want to debug the `com.mycompany.mybeans.AccountBean` class, and the debuggable version of the AccountBean class resides in directory `/QIBM/UserData/WebASadv/enterprisebeans/com/mycompany/mybeans`, specify `-classpath /QIBM/UserData/WebASadv/enterprisebeans` in the **Command line arguments** field. The corresponding Java source files can reside in the same directory as the Java class files, or you can specify the directory containing the source files (as described in the next step).

5. Specify the directory containing your source to be debugged in the `-DDEBUGSOURCEPATH` Java Virtual Machine property for the application server. On the **General** tabbed page of the application server’s properties, add the `-DDEBUGSOURCEPATH` option to the **Command line arguments** field:

   ```
   -DDEBUGSOURCEPATH /qibm/userdata/webasadv/myuserid/mysourcedirectory
   ```
where /qibm/userdata/webasadv/myuserid/mysourcedirectory is the directory that contains the Java source files.

6. Click the Apply pushbutton.

7. Select the Debug tab and ensure that Debug enabled and Object Level Tracing enabled are selected.

8. Click the Apply pushbutton.

For more information on using the WebSphere administrative console, please see the WebSphere Application Server for AS/400 V3.02 documentation.

Setting the OLT viewer monitoring mode
To monitor the execution flow of your web artifact using OLT, do the following:

1. In order to set up the appropriate 'Monitoring Mode' for OLT, ensure that you are using JDK 1.2.2 and enter the following at a command line:

   oltcc
   This will invoke the Object Level Trace Client Controller window.

2. Expand the OLT Client Controller Settings node and select Monitoring Mode in the left part of the window.

3. Select the Trace and debug with prompt radio button in the right part of the window.

4. Click Apply and then Exit.

Starting the OLT viewer
1. Start OLT by doing one of the following:
   - Type OLT at a command line, ensuring that you are using JDK 1.2.2.

2. Select Online Mode in the OLT viewer window Options pull-down menu.
This invokes the OLT Trace page.

3. Ensure that **Step-by-step Debugging Mode** is enabled.

For more information, please see the OLT documentation.

Starting the WebSphere Application Server and invoking the web artifact

1. Start the application server by clicking the **Start** icon in the WebSphere Administrative Console. Once you are notified that the application server has started successfully, click **OK**. At this point, you will see graphs in the OLT viewer, for example:
2. Notice that the application server’s AS/400 job number is shown in the 
   Process ID field.

3. Invoke the web artifact, for example, by a URL:

   ![Image of Web Browser]

   When you invoke the web artifact, you are presented with the Method 
   breakpoints dialog box, containing a list of methods. After selecting the 
   appropriate method, click the OK pushbutton, ensuring first that Step-by-step 
   debugging mode is selected. For example:

   ![Image of Method breakpoints dialog box]

   The IBM Distributed Debugger starts with the web artifact source in the Source 
   pane. You will also notice that the OLT graph now shows an entry point for your 
   artifact. For example:
After you exit from the debugger, you need to exit manually from OLT and the application server.

RELATED CONCEPTS
Debugging servlets, JSP, and EJB

RELATED TASKS
Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP
Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP
Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

RELATED REFERENCES
Preparing servlets, EJB, and JSP for debugging
An EJB example
A servlet example
A JavaServer Page example

Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

When preparing to debug a web artifact, such as a servlet, Enterprise Java Bean (EJB), or JavaServer Page (JSP), begin by compiling the object with the appropriate Java compiler options. For more information on preparing these web artifacts, please see the related topics below.

Once the web artifact is compiled with the appropriate options, you must put the web artifact’s location in the application server CLASSPATH, start the Object Level Trace (OLT) viewer, ensure that Step-by-step Debugging Mode is set, then start the application server, and invoke the web artifact. After you complete these steps, you will be prompted to step into the appropriate method. After selecting this method, clicking OK will launch the debugger which will open at the first executable line in the method.
**Note:** To avoid holding up other AS/400 jobs with the job that is created for debugging, it is recommended that you create a separate application server instance for the debug session and deploy your servlet, JSP, or Enterprise Java Bean into it. This way, you do not impact the execution of other servlets, JSP files, or Enterprise Java Beans. This is accomplished by using a different port number for the debugging session.

To set the CLASSPATH and SOURCEPATH in the WebSphere Application Server for AS/400, do the following:

1. Start the WebSphere Administrative Console through one of the following methods:
   - Select **Programs > IBM WebSphere > Application Server Vxx > Application Server’s Console**, where xx is the application server version number, from the Windows **Start** menu.
   - From a command line, issue the `adminclient <host> <port>` from the `<install-root>`\bin directory, where `<install-root>` is the directory where the WebSphere Administrative Console is installed, `<host>` is the host name of your AS/400 system, and `<port>` is the AS/400 port number that you intend to use.

2. Use the tree to locate and select the application server that will service the web artifact. The settings for the application server appear on the right-hand side of the console.

3. Specify the directory or directories in which the debuggable class files are located, using the `-classpath` Java Virtual Machine property for the application server which will host the debuggable Enterprise Java Beans, servlets or JSP files. On the **General** tabbed page of the application server’s properties, add the `-classpath` option to the **Command line arguments** field:

   `-classpath /qibm/userdata/webasadv/myuserid/mydirectory`

   where `/qibm/userdata/webasadv/myuserid/mydirectory` is the root directory containing the debuggable class files.

   **Note:** In most cases, `myuserid` will be default. For example, if you want to debug the `com.mycompany.mybeans.AccountBean` class, and the debuggable version of the AccountBean class resides in directory `/QIBM/UserData/WebASadv/enterprisebeans/com/mycompany/mybeans`, specify `-classpath /QIBM/UserData/WebASadv/enterprisebeans` in the **Command line arguments** field.
4. Click the **Apply** pushbutton.

5. Select the **Debug** tab and ensure that **Debug enabled** and **Object Level Tracing enabled** are selected.

6. Enter the name of your client or workstation in the **OLT Server Host** field and leave the value that defaults in the **OLT Server Port** as is.

7. Enter the **SOURCEPATH** in the corresponding entry field.

8. Click the **Apply** pushbutton.

For more information on using the WebSphere administrative console, please see the WebSphere Application Server for AS/400 V3.02 documentation.

**Starting the OLT viewer**

To monitor the web artifact with the OLT viewer, do the following:

1. Start OLT by doing one of the following:
   - Type `OLT` at a command line, ensuring that you are using JDK 1.2.2.
   - From VisualAge for Java Enterprise Edition, Version 3.5 or later, select **Tools > Enterprise Toolkit for AS/400 > Start OLT viewer.**
2. Select **Trace and Debug** from the **Execution mode** pull down list and ensure that the **Debugger hostname** field points to your workstation. For example:

3. Click the **Apply** pushbutton.

For more information, please see the OLT documentation.

**Starting the WebSphere Application Server and invoking the web artifact**

1. Start the application server by clicking the **Start** icon in the WebSphere Administrative Console.
2. Notice that the application server's AS/400 job number is shown in the **Process ID** field.
3. Invoke the web artifact, for example, by a URL:

When you invoke the web artifact, you are presented with the **Method breakpoints** dialog box, containing a list of methods. After selecting the
appropriate method, click the **OK** pushbutton, ensuring first that **Step-by-step debugging mode** is selected. For example:

The IBM Distributed Debugger starts with the web artifact source in the Source pane. You will also notice that the OLT viewer now shows the **OLT Trace** page and that the OLT graph now shows an entry point for your artifact. For example:

After you exit from the debugger, you need to exit manually from OLT and the application server.

**RELATED CONCEPTS**
Debugging servlets, JSP, and EJB

**RELATED TASKS**
Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP
Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP
Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging
Preparing servlets, EJB, and JSP for debugging

Follow these steps to enable your web artifacts, such as servlets, Enterprise Java Beans (EJB), or JavaServer Pages (JSP), and application server for debugging:

1. Compile the servlet, EJB, or JSP file with the -g option, for example:
   ```
   javac -g mycompany.mypackage.MyServlet.java
   ```

2. From the AS/400 command line, run the Create Java Program (CRTJVAPGM) command and specify 10 for the optimization level. For example, enter the following command (as one line, with no space between “myservlets/” and “MyServlet.class”):
   ```
   CRTJVAPGM
   CLSF('/QIBM/UserData/WebASAdv/myservlets/com/mycompany/myservlets/MyServlet.class')
   OPTIMIZE(10)
   ```

3. Deploy your servlet, JSP file, or Enterprise Java Bean in an application server if you have not already done so.

An EJB example

The following scenario describes the steps that you can take to prepare an Enterprise Java Bean running in a WebSphere Application Server on the AS/400 for tracing and debugging.

Assumptions:

- You have an Enterprise Java Bean, AccountBean (part of the com.yourcompany.yourbeans package), which you want to debug.
- The source for the bean is located in the /QIBM/UserData/WebASAdv/yourbeans/com/yourcompany/yourbeans directory.
- You have created an application server, DebugServer, through the Administrative Console.
- You have created your Enterprise Java Bean within the DebugServer application server.
Preparation steps

1. Compile the Java code for debugging. If you use VisualAge for Java to develop your Enterprise Java Beans, specify that debugging attributes be included in the class files when you export the beans. Once you have exported the JAR file, extract the Java source files and the debuggable class files to a directory on the AS/400. For the purposes of this documentation, that directory is /QIBM/UserData/WebAsAdv/yourbeans.

If you are not using Visual Age for Java to develop your beans, use the -g option of the javac command when you compile your source. The Java source and debuggable class files can be placed in the same directory or you can place the Java source files in a different directory and specify the location of the source files to the debugger. For example, to compile the AccountBean Enterprise Java Bean, run the following command:

```
javac -g AccountBean.java
```

For the purposes of this scenario, put the Java source file and debuggable class file in the /QIBM/UserData/WebAsAdv/yourbeans/com/yourcompany/yourbeans directory on the AS/400 server.

2. Use the CRTJVPAGM command to set the optimization level to 10 for the debuggable class file. For this scenario, enter the following command from an AS/400 command line:

```
CRTJVPAGM CLSF('/QIBM/UserData/WebAsAdv/yourbeans/com/yourcompany/yourbeans/AccountBean.class') OPTIMIZE(10)
```

**Note:** This command has been wrapped for display purposes. Enter the command as one line, with no space between yourcompany/ and yourbeans.

3. Your EJB is now ready for tracing and debugging.

**RELATED CONCEPTS**
Debugging servlets, JSP, and EJB

**RELATED TASKS**
Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP
Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP
Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging
Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

**RELATED REFERENCES**
Preparing servlets, EJB, and JSP for debugging
A servlet example
A JavaServer Page example

**A servlet example**

The following scenario describes the steps that you can take to prepare a servlet running in a WebSphere Application Server on AS/400 for tracing and debugging.

**Assumptions:**
- You have a servlet, AccountServlet, which is part of the com.yourcompany.yourservlets package that you want to debug. The source for the servlet is located in /QIBM/UserData/WebAsAdv/yourservlets/com/yourcompany/yourservlets.
You have created an application server, DebugServer, using the WebSphere Administrative Console.

You have deployed your servlet within the DebugServer application server.

Preparation steps

1. Compile the Java code for debugging. If you use VisualAge for Java to develop your servlets, specify that debugging attributes be included in the class files when you export the servlet. If you use the javac command to compile your servlet, use the -g option when compiling your source. The Java source and debuggable class files can be placed in the same directory or you can place the Java source files in a different directory and specify the location of the source files to the debugger.

   The Java source and debuggable class files must be placed in the directory on the AS/400 which you specify in the CLASSPATH of the application server. For example, to compile the Account Servlet, specify the following command:

   javac -g AccountServlet.java

   For the purposes of this documentation, put the Java source file and debuggable class file in the
   /QIBM/UserData/WebAsAdv/yourservlets/com/yourcompany/yourservlets directory on AS/400.

   Note: If you are using VisualAge for Java to develop your servlet, be sure to export the Java source file for the servlet as well as the class file.

2. Use the CRTJVAPGM command to set the optimization level to 10 for the debuggable class file. For this scenario, run the following command:

   CRTJVAPGM CLSF('/QIBM/UserData/WebAsAdv/yourservlets/com/yourcompany/yourservlets/AccountServlet.class') OPTIMIZE(10)

   Note: This command has been wrapped for display purposes. Enter the command as one line, with no space between yourcompany/ and yourservlets.

3. Your servlet is now ready for tracing and debugging.

RELATED CONCEPTS
- Debugging servlets, JSP, and EJB

RELATED TASKS
- Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP
- Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP
- Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging
- Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

RELATED REFERENCES
- Preparing servlets, EJB, and JSP for debugging
- An EJB example
- A JavaServer Page example

A JavaServer Page example

The following scenarios describe the steps you can take to prepare the servlet code generated for a JSP running in the WebSphere Application Server environment for debugging. Two scenarios are described - one for debugging a JSP compiled for the JSP 0.91 specification and one for debugging a JSP compiled for the JSP 1.0
specification. The purpose of these scenarios is to provide simple examples of debugging JSP files which you can extrapolate to your own situation.

**Assumptions**
- You have the default administrative server running which has the sample configuration installed.
- The sample configuration is installed by default when you first start your WebSphere Application Server environment.

**Debugging steps**
The steps for debugging a JavaServer Page compiled for the JSP 0.91 specification are somewhat different than the steps for debugging a JSP compiled for the JSP 1.0 specification.

**Debugging a JavaServer Pages 0.91 file**
This scenario describes debugging `simple.jsp` which is installed under the examples Web application when the Sample Configuration has been installed:

1. To cause the code to be generated for the `simple.jsp` file, invoke the JSP from a browser as follows:

   ```
   http://yourAS400:yourPort/webapp/examples/simple.jsp
   ```

   where `yourAS400` is the AS/400 host name, and `yourPort` is the port number.

   **Note:** The default WebSphere Application Server environment must be running with the Default Server application server and your HTTP server instance started as well.

   WebSphere Application Server generates Java code from the JSP file and compiles it dynamically. Once the code has been generated, you can compile it with `debug`. The generated servlets (both source and class files) for JSP files are located in the pagecompile directory, in the host and web application subdirectories of the temp directory for the administrative server, where web application is the name of the web application in which the JSP is deployed and host is the name of the virtual host with which the web application is associated. For example:

   For this example, using the default administrative server directory (`admin.instance.root=/QIBM/UserData/WebASAdv/default`), the servlet for `simple.jsp` in Web application examples under virtual host `default_host` is generated in the following directory:

   `/QIBM/UserData/WebASAdv/default/temp/default_host/examples/pagecompile`

   If the `workingDir` initial parameter is set for the JSP enabler for your Web application, the pagecompile directory and its subdirectories are created under the directory specified for this parameter.

   **Note:** The user profile under which the application server is running must have authority to the directory specified by the `workingDir` parameter. By default, the application server runs under the QEJBSVR user profile.

   The naming convention for the generated files is `_jspFileName_xjsp`. (If your JSP file name contains an underscore, the generated file name contains two underscores in place of each single underscore.) For this example, the JSP file `simple.jsp` results in the generation of `_simple_xjsp.java` and `_simple_xjsp.class`.

   **Note:** The generated servlet class is created as part of the pagecompile package. In other words, the Java source code contains the following package statement:

   ```
   package pagecompile;
   ```
2. Compile the generated Java code for debug by using the `javac -g` command.
To compile the generated Java code for `simple.jsp`, you need to have the following items in your class path:

- `/QIBM/UserData/WebASAdv/default/temp/default_host/examples`
  This directory contains any helper classes generated as a result of the contents of the JSP file.
- `/QIBM/ProdData/WebASAdv/lib/servlet.jar`
  This jar contains the servlet API classes.
- `/QIBM/ProdData/WebASAdv/lib/server/ibmwebas.jar`
  This jar contains the necessary JSP compiler classes.

3. Use the `CRTJVAPGM` command to set the optimization level to 10 for the debuggable class file. For this example, you should issue the following CL command from the AS/400 command line:

```
CRTJVAPGM
CLSF('/QIBM/UserData/WebASAdv/default/temp/default_host/examples/
  pagecompile/_simple_xjsp.class') OPTIMIZE(10)
```

**Note:** This command has been wrapped for display purposes. Enter the command as one line, with no space between `examples/` and `pagecompile`.

4. Your JSP 0.91 is now ready for tracing and debugging.

**Note:** If you modify the JSP file, you need to be sure to stop and restart the application server after performing the steps described above to create the debuggable version in the proper location. Restarting the application server causes the newly generated servlet code to be loaded.

**Debugging a JavaServer Pages 1.0 file**

This scenario describes debugging `very_simple.jsp` which is installed under the default_app Web application when the Sample Configuration has been installed. This scenario involves creating your own Web application and copying the `very_simple.jsp` file to the document root of your Web application.

1. For this example, create your own Web application as follows:
   a. From the Tasks page of the Administrative Console, expand the Configuration tree item.
   b. Select Configure a Web Application and click the Start icon.
   c. Specify JSPDbgTest for the WebAppName, and ensure that the Enable JSP 1.0 option is selected.
   d. Click Next.
   e. Click Next.
   f. Click Finished.

2. To debug the generated servlet code for the JSP file, configure the JSP 1.0 Processor to keep the generated source file:
   a. From the Topology page of the Administrative Console, expand the JSPDbgTest Web application tree item (found under Default Server > servletEngine).
   b. Select the JSP 1.0 Processor tree item.
   c. The property page for the JSP 1.0 Processor is displayed on the right side of the console. Click the Advanced tab of the property page.
   d. In the Init Parameters field, add the parameter `keepgenerated`. Specify true for the Init Parm Value.
   e. Click Apply.

3. Create the appropriate directories for the Web application created above. The directories are created as subdirectories of the hosts/default_host directory in
the root directory for the administrative server (the root directory for the administrative server is specified by the admin.instance.root property in the admin.properties file). For this example, the instance root of the administrative server is /QIBM/UserData/WebASAdv/ default. Create the following directories under /QIBM/UserData/WebASAdv/default/hosts/default_host:
JSPDbugTest JSPDbugTest/web JSPDbugTest/servlets

4. Copy JSP file very_simple.jsp from
/QIBM/ProdData/WebASAdv/hosts/default_host/default_app/web to
/QIBM/UserData/WebASAdv/default/hosts/default_host/JSPDbugTest/web. You are placing the JSP file in the document root of your new Web application so that the JSP file can be invoked from a browser.

5. Start the Default Server application server. From the Topology page of the Administrative Console, right-click the Default Server tree item and click Start. If the Default Server is already started, right-click on the JSPDbugTest Web application, and select Restart Web App.

6. To cause the code to be generated for the very_simple.jsp file, invoke the JSP from a browser as follows:
http://yourAS400:yourPort/webapp/JSPDbugTest/very_simple.jsp

Note: The default WebSphere Application Server environment must be running with the Default Server application server and your HTTP server instance started as well.

As a result of the JSP file invocation, the following directory structure is created:

<admin.instance.root>/temp/default_host/JSPDbugTest/<packageName>

where <admin.instance.root> is the instance root directory for the administrative server, and <packageName> is a directory structure determined by substituting the ‘.’ separator in the package name of the generated Java class with ‘/’ file separator. The package to which the generated Java class belongs is a pseudo mangling of the directory structure and can be determined by displaying the generated .java file and looking at the package statement. See the following information for where the generated .java file can be found. For this example, the following directory structure is created:

/QIBM/UserData/WebASAdv/default/temp/default_host/JSPDbugTest/QIBM/UserData/WebASAdv/d_00025fault/hosts/_00025efault_0005fhost/JSPDbugTest/web

Note: The above path name has been wrapped for display purposes. The directory path does not contain a space between the second UserData/ and the second WebASAdv.

Three files are generated as a result of invoking a JSP using the JSP 1.0 Processor:

- JspFileName.class
- JspFileName.jsp_n.java
- packagenameJspFileName.dat

where JspFileName is the name of the JSP file invoked, n is an integer starting at 1 which is incremented each time the JSP file is invoked after a modification has been made to the JSP file, and packagenameJspFileName is the string formed by prepending the package to which the generated Java class belongs to the JSP file name. The package to which the generated Java class belongs is a pseudo mangling of the directory structure and can be determined by displaying the generated .java file and looking at the package statement.

The files are created in the following directory structure format:
For this example, the following files are generated in the
/QIBM/UserData/WebASAdv/default/temp/default_host/JSPDbgTest directory:

very_simple.class very_simple_jsp_1.java
QIBM.UserData.WebASAdv.d_00025fault.hosts._00025efault_0005fhost.JSPDbgTest.webvery_sim

The package statement in the generated .java file for this example is package
QIBM.UserData.WebASAdv.d_00025fault.hosts._00025efault_0005fhost.JSPDbgTest.web;

7. Copy the packageName directory structure created when you invoked the JSP to
the servlets directory for the Web application so that the following directory
structure is created:
<admin.instance.root>/hosts/default_host/JSPDbgTest/servlets/<packageName>
where <admin.instance.root> is the instance root for the administrative server
and <packageName> is the package name as a directory structure. For this
example, the resulting directory structure is (this includes the full path to the
servlets directory for the Web application):
/QIBM/UserData/WebASAdv/default/hosts/default_host/JSPDbgTest/servlets/QIBM/
UserData/WebASAdv/QIBM/UserData/WebASAdv/d_00025fault/hosts/_00025efault_0005fhost/JSPDbgTest/web

Note: This path has been wrapped for display purposes. The directory path
does not actually have a space between the second QIBM/ and the second
UserData.

8. Copy the generated .java file to the directory created in the previous step. For
this example, copy the very_simple_jsp_1.jsp file from the
/QIBM/UserData/WebASAdv/default/temp/default_host/JSPDbgTest directory to the
following directory:
/QIBM/UserData/WebASAdv/default/hosts/default_host/JSPDbgTest/servlets/QIBM/
UserData/WebASAdv/QIBM/UserData/WebASAdv/d_00025fault/hosts/_00025efault_0005fhost/JSPDbgTest/web

Note: This path has been wrapped for display purposes. The directory path
does not actually have a space between the second servlets/ and the second
QIBM or between the second hosts/ and _0002efault_005fhos.

By default, servlets for the JSPDbgTest Web application are looked for in the
servlets directory and its subdirectories.

To debug the JSP file, you must debug the generated servlet. For JSP files
which use the JSP 1.0 Processor, you need to invoke the generated servlet from
your browser, instead of the JSP file as you are able to do when debugging a
JSP 0.91 file.

9. Compile the generated Java code for debug by using the javac -g command.
To compile very_simple_jsp_1.java, you need to have the following items in
your class path:
   • /QIBM/UserData/WebASAdv/default/temp/default_host/JSPDbgTest
     This directory contains any helper classes generated as a result of the
     contents of the JSP file.
   • /QIBM/ProdData/WebASAdv/lib/servlet.jar
     This jar contains the servlet API classes.
   • /QIBM/ProdData/WebASAdv/lib/server/ibmwebas.jar
     This jar contains the necessary JSP compiler classes.
   • /QIBM/ProdData/WebASAdv/lib/server/jsp10.jar
     This jar contains the IBM JSP 1.0 extensions. It is possible that you will not
     need this jar. For more information about the IBM extensions to the JSP 1.0
     API, see JSP 1.0 programming reference.
10. Use the CRTJVAPGM command to set the optimization level to 10 for the debuggable class file. For this example, run the following CL command from the AS/400 command line:

```cl
CRTJVAPGM
CLSF('/QIBM/UserData/WebASAdv/default/hosts/default_host/JSPDbgTest/
servlets/QIBM/UserData/WebASAdv/d_00025fault/hosts/_00025efault_0005fhost/
JSPDbgTest/web/very_simple_jsp_1.class') OPTIMIZE(10)
```

**Note:** This command has been wrapped for display purposes. Enter the command as one line, with no space between JSPDbgTest/ and servlets or between _00025efault_0005fhost/ and the second JSPDbgTest.

11. Your JSP 1.0 is now ready for tracing and debugging.

**Note:** If you modify the JSP file, you must stop and restart the application server after you perform the steps described above to create the debuggable version in the proper location. Restarting the application server causes the newly generated servlet code to be loaded.

**Related Concepts**
- Debugging servlets, JSP, and EJB

**Related Tasks**
- Setting up WebSphere Application Server V3.02 for debugging servlets, EJB, and JSP
- Setting up WebSphere Application Server V3.5 for debugging servlets, EJB, and JSP
- Setting up the client machine for WAS 3.02 and OLT for servlet, EJB, and JSP debugging
- Setting up the client machine for WAS 3.5 and OLT for servlet, EJB, and JSP debugging

**Related References**
- Preparing servlets, EJB, and JSP for debugging
- An EJB example
- A servlet example

---

**Reference Information**

**Debug data and RPG, C, Cobol, and CL compiler options**

Debug data is the symbolic and line number information contained in the program object that is used by the debugger.

If you want to debug your application, you must first compile your modules or programs to include debug data.

<table>
<thead>
<tr>
<th>Language</th>
<th>Compile using...</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPM COBOL or RPG</td>
<td>either the *SRCDBG or *LSTDBG option.</td>
</tr>
<tr>
<td>OPM CL</td>
<td>the *SRCDBG option.</td>
</tr>
<tr>
<td>ILE COBOL, CL, or RPG</td>
<td>the *SOURCE, *LIST, *STMT, or *ALL option. The default is *STMT (Statement view).</td>
</tr>
<tr>
<td>ILE C</td>
<td>the *SOURCE, *LIST, *STMT, or *ALL option. The default is *NONE (no debug data).</td>
</tr>
</tbody>
</table>
See your compiler documentation for more details on these compiler options.

**RELATED CONCEPTS**
Optimized code debugging

**RELATED TASKS**
Writing a program for debugging
Compiling a program for debugging

**RELATED REFERENCES**
ILE C++ compiler options
Java compiler options

### ILE C++ compiler options
To create debug data in your C++ program, compile your code with one of the following options to request one or more debug views:

<table>
<thead>
<tr>
<th>Compiler option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Ti+</td>
<td>Compiles your program to produce a module that includes a Source view, a Listing view, and a Statement view for debugging purposes.</td>
</tr>
<tr>
<td>/Til</td>
<td>Compiles your program to produce a module that includes a Listing View and a Statement View for debugging purposes.</td>
</tr>
<tr>
<td>/Tis</td>
<td>Compiles your program to produce a module that includes a Source View and a Statement View for debugging purposes.</td>
</tr>
<tr>
<td>/Tin</td>
<td>Compiles your program to produce a module that includes only a Statement View for debugging purposes.</td>
</tr>
</tbody>
</table>

**RELATED CONCEPTS**
Optimized code debugging

**RELATED TASKS**
Writing a program for debugging
Compiling a program for debugging

**RELATED REFERENCES**
Debug data and RPG, C, Cobol, and CL compiler options
Java compiler options

### Java compiler options
Compile your java source filename.java with the -g option to create debug data for the resulting filename.class. With this option, you can perform all debug functions.

If you do not use the -g compiler option, the debugger will not have access to symbolic information. In this case, the debugger will still be able to display a Source view, if it has access to the filename.java source and to step through your Java class, but you will not be able to view the content of any variables.
Note: Java programs that are compiled with the -O option cannot be debugged.

**RELATED CONCEPTS**
Optimized code debugging

**RELATED TASKS**
Writing a program for debugging
Compiling a program for debugging

**RELATED REFERENCES**
Debug data and RPG, C, Cobol, and CL compiler options
ILE C++ compiler options
Creating a Java program from Java class

Creating a Java program from a Java class
Use the CRTJVAPGM command to create an AS/400 Java program from a Java class file. Specify CRTJVAPGM CLSF(xxxxxx) OPTIMIZE(10), where xxxxxx is the class file. The resulting Java program contains machine-instruction sequences that are run when the Java program is invoked. The Java program is created at optimization level 10 to allow debugging.

If, at startup, the debugger does not find a Java program associated with the Java class file, it directs the Java Virtual Machine (JVM) to create it.

**RELATED TASKS**
Compiling a program for debugging

**RELATED REFERENCES**
Java compiler options

Debugger environment

**Debugger environment settings**
The debugger uses the following environment settings:

- Session Settings
  - Host Name
  - Source Path
  - Port Number
  - Update Production Files
- JVM Creation Settings
  - Environment Variables
  - JNI Library List
  - Properties
- Batch Job Creation Settings
  - Environment Variables
  - Library List

You can change these settings through the Debugger Settings dialog box which opens when you click on Advanced in the Load Program dialog box. For more information about adding, changing or deleting debugger environment settings, see the related tasks below.
**Related Tasks**
- Setting environment variables
- Setting the Java CLASSPATH

**Related References**
- Debugger environment default settings

**Debugger environment default settings**
The settings that define the debug environment are stored in the configuration file CTTLDE.CFG of each product that ships the debugger.

The table below shows the default values of the debugger environment settings:

<table>
<thead>
<tr>
<th>Environment Setting</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Name</td>
<td>None</td>
</tr>
<tr>
<td>Source Path</td>
<td>None</td>
</tr>
<tr>
<td>Port Number</td>
<td>3001</td>
</tr>
<tr>
<td>Update Production Files</td>
<td>Not selected</td>
</tr>
<tr>
<td>JVM Creation Settings</td>
<td>CLASSPATH \QJAVA\</td>
</tr>
<tr>
<td>JVM Environment Variables</td>
<td>None</td>
</tr>
<tr>
<td>JNI Library List</td>
<td>None</td>
</tr>
<tr>
<td>JVM Properties</td>
<td>None</td>
</tr>
<tr>
<td>Batch Job Creation Settings</td>
<td>None</td>
</tr>
<tr>
<td>Batch Job Creation Settings Environment</td>
<td>None</td>
</tr>
<tr>
<td>Variables</td>
<td>None</td>
</tr>
<tr>
<td>Batch Job Creation Settings Library List</td>
<td>None</td>
</tr>
</tbody>
</table>

You can add, change, or delete environment settings through the Debugger Settings dialog box. that opens when you click on Advanced in the Load Program dialog box.

**Related References**
- Debugger environment settings

**Environment variables**

**CLASSPATH environment variable**

The CLASSPATH environment variable tells the debugger, as well as the Java Virtual Machine and other Java applications, where to find your classes on the AS/400.

This variable must be set correctly for any of your Java applications to work.

**Related Tasks**
- Setting environment variables for the debugger
- Setting the Java CLASSPATH
Stepping

Step commands
You can use **step commands** in the Source pane to step through your program a single statement at a time.

The following types of step commands are available:

<table>
<thead>
<tr>
<th>Step Command</th>
<th>Button</th>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step Over</td>
<td><img src="image" alt="Button" /></td>
<td>F10</td>
<td>Executes the current statement, without stopping in any program or procedure called within the statement.</td>
</tr>
<tr>
<td>Step Debug</td>
<td><img src="image" alt="Button" /></td>
<td>F7</td>
<td>Executes the current statement. Execution stops at the next statement encountered for which debug information is available. This could be in the current procedure, in the called procedure, or in a procedure called within the called procedure.</td>
</tr>
</tbody>
</table>

Program execution may stop earlier than indicated in the step command descriptions, if the Distributed Debugger encounters a breakpoint or an exception occurs.

If you issue a step command while debugging a multithreaded application, execution may stop in a different thread.

**RELATED TASKS**
Stepping through a program

**RELATED REFERENCES**
Stepping and procedures

**Stepping and procedures**
In a source code line that contains multiple procedure calls, you can choose to step over all the calls, step through the calls individually, or run to a location in the program. Given a complex C++ call such as `func1(func2(), func3());`, you can do the following:

- Step over the entire line with a single **Step Over** command.
- Step into each called procedure for which debug information is available, with a series of **Step Debug** commands. Each time you use Step Debug to step into such a procedure, you can then step through the procedure. The debugger steps over any procedure for which debug data is not available, such as library and system functions.
• Run to a specific source code line number in the call. When running to a location, the program will run to a selected statement unless an active breakpoint is hit, an exception occurs or the end of the program is reached. The program will run to the statement before executing it or any of its procedure calls. For more information on running to a location in a program, see the related topic below.

**RELATED TASKS**
Running to a location
Stepping through a program

**RELATED REFERENCES**
Step commands

**Program name specifications**

When attaching to an existing AS/400 job, you can enter the program name using one of the following formats:

• Library/program — The debugger searches the specified library for the program. If the program is not found in the specified library, an error dialog is displayed.

• *LIBL/program — All of the libraries in the library list for the job specified in the **Job to debug** entry field of the Load Program dialog box are searched until the first match for the specified program name is found. If the program is not found in the library list, an error dialog is displayed.

• *CURLIB/program — The current library for the job specified in the **Job to debug** entry field is used to locate the program. If you do not have a library designed as the current library, QGPL is searched instead. If the program is not found, an error dialog is displayed.

• Program — When only the program name is specified, the search path used is the same as that used for *LIBL/program.

• Class name.

When you load a program without specifying a job, you can enter the program name using one of the following formats:

• Library/program — The debugger searches the specified library for the program. If the program is not found in the specified library, an error dialog is displayed.

• Class name.

**RELATED TASKS**
Starting the debugger
Loading a program without specifying a job from a command line
Loading a program without specifying a job from the debugger user interface
Attaching to an existing AS/400 job from a command line
Attaching to an existing AS/400 job from the debugger user interface

**Job name specifications and getting a job list**

When specifying a job for debugging, a fully qualified AS/400 job name has the format **job/user/number**, where:

• **job** is the name of the job as identified to the system, for example, QPADEV0017.

• **user** is the user profile under which the job is running, for example, MYUSER.

• **number** is the system-assigned job number, for example, 001234.
If the system-assigned job number is known, then specifying it in the format //number will specify a unique job and specification of job and user will not be necessary.

In the Load Program dialog box, you can use one of the following methods to retrieve a subset list of the jobs available on the AS/400 system. Make the following entries in the Job to debug field and click the Job List push button to invoke the Job List dialog box:

- Entering QPADEV0017// gives you a list of jobs with job name QPADEV0017.
- Entering /MYUSER/ gives you a list of jobs with user profile MYUSER.
- Entering //001234 gives you the job assigned this job number.

You can also specify an abbreviated job name. The following formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*, *ALL, blank</td>
<td>All occurrences are found.</td>
</tr>
<tr>
<td>abc*</td>
<td>All occurrences starting with the character string abc are found.</td>
</tr>
<tr>
<td>abc</td>
<td>Only exact matches of the character string abc are found.</td>
</tr>
</tbody>
</table>

The following examples illustrate how to use these formats:

- Entering QPADEV0017/*ALL/ gives you a list of all the jobs in QPADEV0017.
- Entering // or leaving the Job to debug field blank gives you a list of all the jobs on the system.
- Entering /G*/ gives you a list of all the jobs that are owned by user IDs that start with a G.
- Entering G*/*/ gives you a list of all the jobs owned by anyone in a job that starts with a G.

If a wild card indicates a user ID that owns a single job, then specifying the wild card in this field will cause that job to be used. However, using wild cards to indicate multiple jobs is useful only to filter the list of jobs when the Job List push button is selected. When you select the Load push button, exactly one job must be indicated or an error dialog is displayed.

**Related Tasks**
- Starting the debugger
- Attaching to an existing AS/400 job from the debugger user interface

**Job types**

Jobs that are running on the AS/400 can be one of the following:

- Interactive jobs, which are enabled to run an application with screen input/output.
- Batch jobs.
- Batch immediate (BCI) jobs, which are enabled to run a threaded application. Applications with screen input/output should not be run in a BCI job.

All types of jobs can be attached to by the debugger. When you load a non-Java application without specifying a job, a BCI job is automatically created on the AS/400. When you load a Java class without specifying a job, a Java Virtual Machine (JVM) is created on the AS/400.
Starting the debugger
Attaching to an existing AS/400 job from the debugger user interface

Authorities required for debugging
The user profile that you use to sign on to an AS/400 system in the debugger Logon dialog must have the following authorities:
• *USE authority to the Start Debug (STRDBG) command
• *USE authority to the End Debug (ENDDBG) command
• *USE authority to the Start Service Job (STRSRVJOB) command
• *USE authority to the End Service Job (ENDSRVJOB) command
• Either *CHANGE authority to the program or Java class being debugged, or
  *USE authority to the program being debugged and *SERVICE special authority.

If the job that you are debugging is running under a different user profile than the user profile you use to sign on to the AS/400 system from the debugger, the user profile that you use to sign on to the AS/400 system from the debugger must have the following authorities:
• *USE authority to the user profile that the job you are debugging is running under.
• *JOBCTL special authority if you do not explicitly use fully qualified program names (library/program). In other words, if you use *CURLIB or *LIBL or you do not specify a library name, you need to have *JOBCTL special authority.

The group profile QPGMR gives you the correct authority to the STRDBG, ENDDBG, STRSRVJOB, and ENDSRVJOB commands and *JOBCTL special authority.

Preparing for debugging
Starting the debug server

Debugger limits
The following limits apply when debugging your application:
• The largest string that can be displayed is 4080 characters.
• The largest number of elements displayed in a COBOL array or an RPG array is 500.
• The largest number of fields displayed in a COBOL record or an RPG structure is 500.
• Only 128 characters per line are displayed for the source views retrieved from the AS/400. Files residing on the PC will have the entire line displayed regardless of the line length.
• Source physical files that have a record size greater than 240 characters cause a message to be written to the job log for each record read. (The job log that the messages are written to is the job log of the debug server job that is serving your debug session.) This behavior slows down processing and may cause your debug session to end if the job log grows too large.
• Local variables for ILE RPG and COBOL cannot be displayed in the Locals pane.

Debugger performance considerations

RELATED TASKS
Starting the debugger
Attaching to an existing AS/400 job from the debugger user interface

RELATED TASKS
Preparing for debugging
Starting the debug server

RELATED CONCEPTS
Debugger performance considerations
Debugger performance considerations

To get optimal performance from the debugger, consider the following points:

**Expression Evaluations**
- Complex expressions take longer to evaluate than simple expressions. Performance is only an issue when you are monitoring an expression, since the expression must be evaluated each time the debugger stops.
- The settings of the Default Monitor Representations affect the performance of expression evaluation:
  - Representing character pointers, arrays, and character arrays as hexadecimal pointers gives the best performance.
  - Representing structures using System Defaults performs better than using User Defaults.
  - Representing a character array as a string is faster than representing it as an array.
- Evaluating all of the elements of a large array takes longer than evaluating single elements. Use the Monitors pane to evaluate a single element.

**Step Performance**
Step performance is affected by the number of enabled variables or expressions in the monitors and the complexity of the expression. Step performance can be improved by:
- Disabling or deleting expressions that no longer need to be monitored
- Displaying only single elements of an array.
- After following a chain of pointers to a variable, disabling the pointers used and leaving only the variable active in the monitor.
- Completely collapsing the threads in the Locals pane.

**Avoid expanding the procedures within a module**
This function requires a lot of interaction with the AS/400. If a large number of methods or procedures are in a module, searching for the method or procedure name by using the Find Method or Find Procedure choice on the Source menu is faster.

**Using PC files instead of AS/400 source members**
For non-C++ or Java programs, performance can be improved by copying the files to the PC and using the Change text file choice from the Source menu to specify the path name of the PC file. This technique is especially useful when debugging from remote sites.

**Searching for a string in Source view or Listing view**
String searches can be speeded up by the following:
- Keeping the source file on the workstation.
- Using the Find Method or Find Procedure choice to search for methods or procedures.
- Searching the Listing view instead of a Source view that is on the AS/400.

**Using the Optional Parameters entry fields on line breakpoints**
Values specified for these options will significantly slow down your program, because the debugger must stop for the breakpoint and evaluate the From/To/Every clause each time. Even though you do not see the program stop, it is in fact stopping so that the debugger can evaluate the stop conditions.
If possible, an alternative is to set a conditional breakpoint by specifying an expression.

**Avoid setting a large number of watches**
When a watch is set, the system checks after each instruction whether the value of the watched variable or expression has changed. Setting many watches may lead to slower performance.

**RELATED TASKS**
- Changing the representation of monitor contents
- Setting a line breakpoint
- Stepping through a program

**RELATED REFERENCES**
- Debugger limits

**Expressions qualified to thread number**
Within a program, every expression is qualified to a specific thread.

If you enable Tool Tip Evaluation for variables, the thread number that a variable is qualified to is displayed in the pop-up. For information on enabling Tool Tip Evaluation, see the related topic below.

**Note:** A variable with the same name (but belonging to different scopes) may have different values on the same thread.

**RELATED TASKS**
- Enabling and using Tool Tip Evaluation for variables
- Viewing the contents of a variable or expression

**Problems getting a Source or Listing view**
If you are in the Statement view of a section of code, you may find that you cannot obtain a Source or Listing view of your code. There are several likely reasons for such problems:

- The code you are debugging was not compiled with the debug options to generate a Source or a Listing view.
- The debugger is finding a different version of the program than the one you compiled with the appropriate options.
- The code you are debugging was compiled with the appropriate debug information, but the debugger cannot locate the source code. If you try to switch to a Source or Listing view in this situation, the Source Filename dialog box opens so that you can enter the path of the source file. If you select **Cancel**, the view remains a Statement view.
- The code you are debugging was not compiled with the appropriate compiler options to generate a Source or Listing view, because it is not code you wrote, but code in a service program or other program that your program uses.

**RELATED CONCEPTS**
- Source views

**RELATED TASKS**
- Changing views in the Source pane
- Locating source code
Values that are valid for the current representation

When you are entering a value in an entry field in one of the monitors, such as the contents of a variable, the debugger checks that the value you enter is valid for the current representation of that entry field.

A value is valid for the current representation if it contains only the characters used for that representation, and does not exceed the length of the variable involved. For example, if you want to change the contents of a character string and the current representation is as a text string, you must enter a new string in double quotes, and the length of the string must not exceed the declared string size.

Related tasks
Changing the contents of a variable or expression

Previous load program methods

Startup controls and control functionality in previous versions of the AS/400 debugger that have changed to those in the Distributed Debugger Load Program dialog box as follows:

<table>
<thead>
<tr>
<th>Previous debugger version startup control...</th>
<th>Is now...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocation Mode</td>
<td>Specify the dominant language the debugger will use</td>
</tr>
<tr>
<td>Three invocation modes available, including:</td>
<td>Invocation modes are mapped onto dominant language/job name combinations. The dominant languages supported are RPG, COBOL, CL, Java, C, and C++. The dominant language determines the terminology used by the debugger.</td>
</tr>
<tr>
<td>• Default, for debugging RPG, COBOL, CL, and regular C and C++ programs.</td>
<td></td>
</tr>
<tr>
<td>• Java, for debugging Java AS/400 classes.</td>
<td></td>
</tr>
<tr>
<td>• Non-Java Threaded, for debugging C and C++ threaded applications.</td>
<td></td>
</tr>
<tr>
<td>Program</td>
<td>Specify the name of the program to add to debug</td>
</tr>
<tr>
<td>Specifies the name of the initial program or Java class for the debug session. This field is mandatory.</td>
<td>Specifies the name of the initial program or Java class for the debug session. This field is no longer mandatory.</td>
</tr>
<tr>
<td>Options</td>
<td>Advanced</td>
</tr>
<tr>
<td>Invokes the Debugger Environment Options notebook.</td>
<td>Invokes the Debugger Settings dialog box.</td>
</tr>
</tbody>
</table>

Related tasks
Starting the debugger
Loading a program without specifying a job from the debugger user interface
Notices

Note to U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director  
IBM Canada Ltd.  
1150 Eglinton Avenue East  
Toronto, Ontario M3C 1H7  
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM’s application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1997, 2000. All rights reserved.
Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

**Warning:** Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.
Trademarks and service marks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

- AIX
- AS/400
- DB2
- CICS
- CICS/ESA
- IBM
- IMS
- Language Environment
- MQSeries
- Network Station
- OS/2
- OS/390
- OS/400
- RS/6000
- S/390
- VisualAge
- VTAM
- WebSphere

Lotus, Lotus Notes and Domino are trademarks or registered trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Enterprise Console and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

ActiveX, Microsoft, SourceSafe, Visual C++, Visual SourceSafe, Windows, Windows NT, Win32, Win32s and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Intel and Pentium are trademarks of Intel Corporation in the United States, or other countries, or both.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.