IBM® VisualAge® for Java™, Version 3.5

Distributed Debugger for Workstations

Note!

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

Edition notice

This edition applies to Version 3.5 of IBM VisualAge for Java and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Distributed Debugger

The Distributed Debugger is a client/server application that enables you to detect and diagnose errors in your programs. This client/server design makes it possible to debug programs running on systems accessible through a network connection as well as debug programs running on your workstation.

The debugger server, also known as a *debug engine*, runs on the same system where the program you want to debug runs. This system can be your workstation or a system accessible through a network. If you debug a program running on your workstation, you are performing *local debugging*. If you debug a program running on a system accessible through a network connection, you are performing *remote debugging*.

The Distributed Debugger client is a graphical user interface where you can issue commands used by the debug engine to control the execution of your program. For example, you can set breakpoints, step through your code and examine the contents of variables. The Distributed Debugger user interface lets you debug multiple applications, which may be written in different languages, from a single debugger session. 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 being debugged. The type of information displayed depends on the debug engine that you are connected to.

Each program page is divided into different 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. The types of panes and monitors available on a program page depend on the program you are debugging.

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
Remote debugging

Distributed Debugger: Panes

The types of panes displayed when debugging a program depend on the programming language used. 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 node in a tree list. Expanding a node will display the names of active functions for that thread. The thread your are debugging is highlighted.

Breakpoints pane

The Breakpoints pane contains a view of information about the breakpoints you have set in the program you are debugging. Use the Breakpoints pane to view breakpoints set in your program, modify their properties, enable or disable breakpoints, delete them, or add new ones.

Source pane

The Source pane provides a view of the source code for the program you are debugging. If your program was compiled with debugging information, you have three choices as to how to view it: by its source code, its disassembled machine code, or a combination of the two. To view source code, the source code must be accessible from your workstation, either on a local or a network drive. If the source code file is not found, only a disassembled machine code view is available.

JAVA When debugging interpreted Java classes, only a source code view is available.

Modules pane

C C++ JAVA This pane is displayed when debugging C, C++, or High Performance Compiled (HPC) Java programs.

The Modules pane displays a list of modules loaded while running your program. The items in the list can be expanded to show compile units, files and functions.

Packages pane

JAVA This pane is displayed when debugging interpreted Java programs.

The Packages pane displays a list of the Java packages loaded while running your program. The items in the list can be expanded to show class files, Java source files, and methods.

The remaining panes are monitor panes. For more information on monitor panes, see the related topic below.

RELATED CONCEPTS
Distributed Debugger: Monitors
Distributed Debugger: Overview

Distributed Debugger: Monitors

Depending on the language you are debugging, the Distributed Debugger provides you with monitors to monitor various aspects of your program. The following monitors are available in the Distributed Debugger user interface:

Variables and Expressions (Monitors pane)

The Monitors pane shows variables and expressions that you have selected to monitor. You can enter the variables or expressions in a dialog box or select 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 to change the representation of values.

Tip: Enabling hover help 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 hover help for variables 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 each thread. The Locals pane is updated after each Step or Run command to show what variables are currently in scope and the contents of those variables. It is also used to modify the content of variables or to change the representation of values.

Registers (Registers pane)

The Registers pane allows you to view and change the contents of processor registers for the threads in your program. Although threads may share the same set of registers, the operating system saves the register contents of each thread as the thread is suspended, and restores that thread's processor contents when the thread resumes. The registers are categorized, so you only need to expand the category of registers that you wish to view.

JAVA The Registers pane is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

Storage (Storage pane and Storage Monitors pane)

Storage pane and Storage Monitors pane let you view and change the contents of storage areas used by your program. You can also change the address range to view and modify the contents of storage, and change the representation the debug engine uses to display storage.

The initial Storage pane shows the storage areas used by your program at its starting address.

You can add additional Storage Monitor panes that start at the address of storage allocated to a register, variable, array, class object or expression.

JAVA Storage pane and Storage Monitor panes are not available when debugging interpreted Java programs. They are available when debugging HPC Java programs.

RELATED CONCEPTS
Distributed Debugger: Overview
Distributed Debugger: Panes

RELATED TASKS
Enabling hover help for variables

Chapter 2. Preparing a program for debugging

Writing a program for debugging

You can make your programs easier to debug by following these simple guidelines:

- Do not hand-tune your source code for performance until you have fully debugged and tested the untuned version. Hand-tuning may make the logic of your code harder to understand.
- Where possible, do not put multiple statements on a single line, because 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 to make it easier to verify intermediate results by monitoring the temporary variables.

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.

Recursion and debugging

Recursion does not have to involve a routine calling itself directly; for example: FUNC1 calls FUNC2 calls FUNC3 calls FUNC1. After the call to FUNC3, each time you step into one of these routines, the entry for that call shows a recursion count one higher than the previous entry for that call on the Stacks pane.

You can use the recursion value in the stack frame properties box to detect unintentionally recursive calls.

Limits to debugging recursive function calls

Only the copy of the variables from the most recent invocation of a function can be monitored. Variables from previous invocations of the recursive function cannot be monitored.

UNIX® call handling during debugging

exec() handling

Restriction: AIX This is supported on AIX®

When a process calls exec() , a new program is loaded to replace the current program.

The debugger suspends program execution at this point and opens a dialog similar to the Load Program dialog, which allows you to choose whether to debug program initialization and whether to use a program profile. The name of the new program is shown, but unlike in the Load Program dialog, you cannot change the name. After you select **OK**, the debugger stops at the first instruction of the new program's runtime (if you asked to debug program initialization), or at the first instruction or statement in the new program.

If you checked **Use program profile** or used the p+ option of the idebug command and the Distributed Debugger finds enabled breakpoints in the profile, these breakpoints are set when you start debugging the child process.

RELATED REFERENCES
fork() handling
system() handling

fork() handling

AIX **Restriction:** This is supported on AIX only.

When a process calls `fork()`, an exact copy of that process is created. The process that forked is called the parent, and the new process is called the child. If a process being debugged forks, the Distributed Debugger stops both the parent and child processes, and opens a dialog box that lets you choose whether to continue debugging the parent process or switch to the child process.

Whichever choice you make (**Parent** or **Child**), the Distributed Debugger ignores the process you did *not* choose, and allows it to continue running. Breakpoints set in the process you did not choose are ignored, and the page pertaining to that process is closed. Execution stops at the next source code statement in the program that contains debugging information.

If the process you did *not* choose performs an `exec()`, a new Distributed Debugger page will open for the new child process.

RELATED REFERENCES
exec() handling
system() handling

system() handling

AIX **Restriction:** This is supported on AIX only.

When a program running in a UNIX environment starts another program using a call to `system()`, the `system()` function calls both `fork()` and `exec()`. The following describes the Distributed Debugger's behavior after you perform a **Step Over** command on a line containing a `system()` call, and tells you what actions you should take to begin debugging the child process.

1. The `system()` function calls `fork()`. The Distributed Debugger stops execution and raises a Process fork action dialog.
2. At this point you should choose to debug the child process. Once the Process fork action dialog closes, issue the **Run** command to continue debugging the child process.
3. The new child process calls `exec()` to load `/bin/sh`, and the debugger opens a New process dialog and the active Distributed debugger Source pane shows a disassembly view of the initial runtime entry point of `/bin/sh`.
4. Click **OK** to start debugging the child process.
5. The Distributed Debugger stops in the main function of `/bin/sh`.
6. Issue the **Run** command.

7. The `/bin/sh` process issues a `fork()` call. Again, the Distributed Debugger stops and brings up a Fork action dialog.
8. At this point you should choose to debug the **Child** process. Once the Process fork action dialog closes, issue the **Run** command to continue debugging the child process.
9. The new child process calls `exec()` to load the program specified in the call to `system()` in the original program. The Distributed Debugger opens a New process dialog and the active Distributed Debugger Source pane shows a disassembly view of the initial runtime entry point of the program specified in the call to `system()`.
10. Click **OK**. The Distributed Debugger stops at `main()`. From here you can continue debugging.

RELATED REFERENCES
exec() handling
fork() handling

Compiling a program for debugging

In order to debug your program at the source code level, you need to compile your program with certain compiler options that instruct the compiler to generate symbolic information and debug hooks in the object file. See your compiler reference documentation on how to compile your program with debug information.

RELATED TASKS
Optimized code debugging
Writing a program for debugging

Optimized code debugging

Problems that only surface during optimization are often an indication of logic errors or compile errors that are exposed by optimization, for example using a variable that has not been initialized. If you encounter an error in your program that only occurs in the optimized version, you can usually find the cause of the error using a binary search technique to find the failing module:

1. Begin by optimizing half the modules and see if the error persists.
2. After each change in the number of optimized modules, if the error persists, optimize fewer modules; if the error goes away, optimize more modules. Eventually you will have narrowed the error down to a single module or a small number of modules.
3. Debug the failing module. If possible, turn off the instruction scheduling optimizations for that module. Look for problems such as reading from a variable before it has been written to, and pointers or array indices exceeding the bounds of storage allocated for the pointer or array.

When you debug optimized code, information in debugger panes may lead you to suspect logic problems that do not actually exist. You should bear in mind the points below:

Local variables are not always current

Do not rely on the Local variables monitor to show the current values of variables. Numeric values, character values and pointers may be kept in processor registers.

In the optimized program, these values and pointers are not always written out to memory; in some cases, they may be discarded because they are not needed.

Static and external variables are not always current

Within an optimized function, the values of static or external variables are not always written out to memory.

Registers and Storage monitors are always current

The Registers and Storage monitors are correct. Unlike a monitor that shows actual variables, such as the Locals Variables monitor, the Registers and Storage monitors are always up-to-date as of the last time execution stopped.

Source statements may be optimized away

Use the disassembly view or mixed view to see the source for your program. You may find, for example, that an assignment to a variable in your source code does not result in any disassembly code being produced; this may indicate that the variable's value is never used after the assignment.

Chapter 3. Starting the Distributed Debugger

Setting environment variables for the debugger

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.

You may want to set environment variables for the debug engine and Distributed Debugger. You can set the environment variables based on your operating system. For instructions on setting environment variables refer to your operating system manuals.

RELATED REFERENCES
DER_DBG_CASESENSITIVE
DER_DBG_LOCAL_PATH
DER_DBG_NUMBEROFELEMENTS
DER_DBG_OVERRIDE
DER_DBG_PATH
DER_DBG_TAB
DER_DBG_TABGRID

OS/2 DPATH
INCLUDE
LIBPATH
PATH

JAVA CLASS PATH

Starting the debugger for local debugging

Restriction: Local debugging is supported on AIX and Windows® only.

To start debugging a program locally from the command line, issue the `idebug` command with local debug parameters at a command line prompt.

If you issue the `idebug` command without any options, the debugger will prompt you for the required information in the Load Program dialog.

JAVA **Tip:** Before debugging your interpreted Java program, set your CLASSPATH environment variable to point at all classes and packages you will need to debug your program. Also, ensure that the source code to these classes and packages is accessible from your local system.

Tip: If you have debugged a specific program before and do not want to use the previous profile options, make sure that **Use program profile** is not selectedt or use the `-p-` option of the `idebug` command.

Once the debugger user interface is running, you can debug other programs using the same debugger session by selecting **File > Load Program** from the main menubar.

RELATED TASKS

SOLARIS Starting the debugger on Solaris

RELATED REFERENCES

idebug command

Starting the debugger for remote debugging

The debugger allows you to run the debugger user interface and the debug engine on separate machines. These separate machines can be running different operating systems. When you start the debugger for remote debugging, you first start a debug engine daemon. This daemon waits for a connection from the debugger user interface. Once a connection is established, you can begin to debug your program.

Debugging compiled languages remotely

To start debugging a remote program from the command line:

1. On the remote system, start the debug engine daemon with the `irmtdbgc` command at a command line prompt. If you issue this command without any parameters, you will be prompted for required information in the Load Program dialog on the local system. For information on `irmtdbgc` command parameters, see the related topic below.
2. On the local system, start the debugger user interface with the `idebug` command using the remote debug parameters at a command line prompt. You must specify the `-qhost` parameter and the `-qlang` parameter for the language you are debugging. For information on the `idebug` command parameters, see the related topic below.

Tip: The debug engine is terminated if the debugger cannot load the program you want to debug. Also, the debug engine is terminated when the program you are debugging runs to completion or is terminated manually. To prevent the debug engine from being terminated in these situations, use the `-qsession=multi` option of the `irmtdbgc` command.

Debugging interpreted Java programs remotely

To start debugging a remote interpreted Java program from the command line:

1. On the remote system, set the CLASSPATH environment variable to point at all classes and packages you will need to debug your program.
2. On the remote system, start the debug engine daemon with the `irmtdbgj` command at a command line prompt. If you issue this command without any parameters, you will be prompted for required information in the Load Program dialog on the local system. For information on the `irmtdbgj` command parameters, see the related topic below.
3. On the local system, start the debugger user interface with the `idebug` command with the remote debug parameters at a command line prompt. You must specify the `-qhost` parameter and the `-qlang=java` parameter. For information on the `idebug` command parameters, see the related topic below.

SOLARIS **Restriction:** The above method does not apply when debugging interpreted Java programs on Solaris. To debug an interpreted Java program running on Solaris, see the related topic below.

Warning: When debugging your interpreted Java program remotely, communication between the debugger and the program being debugged may be terminated prematurely by the JVM. If this happens, you will need to reload your program. To reload your program, select **File > Load Program** and enter the required information in to the **Load Program** dialog.

Tip: The debug engine is terminated if the debugger cannot load the program you want to debug. Also, the debug engine is terminated when the program you are debugging runs to completion or is terminated manually. To prevent these situations use the `-multi` option of the `irmtdbgj` command.

RELATED TASKS

SOLARIS Starting the debugger on Solaris

RELATED REFERENCES

JAVA irmtdbgj command
irmtdbgc command
idebug command

Remote debugging

Debugging a program running on one system while controlling the program from another system is known as remote debugging. The debugger supports remote debugging by allowing you to run the debugger user interface on one system, while running the debug engine on another system. The system running the debugger user interface is known as the *local* system. The system where the debug engine runs is known as the *remote* system.

There are two types of remote debugging: *homogeneous* and *heterogeneous*. Homogeneous remote debugging occurs when both the local and remote system use the same operating system. Heterogeneous remote debugging is when the local and remote systems use different operating systems.

When debugging a program remotely, you can start the debugger in one of two ways:

- Start a debug engine daemon, then start the debugger user interface.
- Start a debugger user interface daemon, then start a debug engine.

In both cases, a daemon will listen for a connection. Once a connection is made you can begin to debug your program.

Why use remote debugging
You might want to use remote debugging for the following reasons:

- *The program you are debugging is running on another user's system, and is behaving differently on that system than on your own.*

  You can use the remote debug feature to debug this program on the other system, from your system. The user on the system running that program interacts with the program as usual (except where breakpoints or step commands introduce delays) and you are able to control the program and observe the program's internal behavior from your system.

- *You want to debug a program that uses graphics or has a graphical user interface.*

It is easier to debug an application that uses graphics or has a graphical user interface when you keep the debugger user interface separate from that of the application. Your interaction (or another user's interaction) with the application occurs on the remote system, while your interaction with the debugger occurs on the local system.

- *The program you are debugging was compiled for a platform that the debugger user interface does not run on.*

  You can use the remote debug feature to take advantage of the debugger user interface while debugging the remote application.

390 400 **Restriction:** This information applies to remoted debugging between workstation platforms only. For information for debugging an OS/390® or AS/400® program from a workstation, see the online for help for the Distributed Debugger shipped with products that support OS/390 or AS/400.

RELATED TASKS
Starting the debugger for remote debugging
Starting the debugger user interface daemon

RELATED REFERENCES
Remote debugging limits

Starting the debugger on OS/2®

Restriction: Only remote debugging of OS/2 programs is supported.

To start debugging a program on OS/2 from the command line:
1. On the OS/2 system, issue the `irmtdbgc` or `irmtdbgj` command at a command prompt. If you issue this command without any parameters, you will be prompted for the required information in the Load Program dialog on the local system.
2. On the local system, issue the `idebug` command with the remote debug parameters at a command line prompt. You must specify the `-qhost` parameter.

The `idebug` command must be issued with `-qhost` option when debugging a remote program.

JAVA **Tip:** When debugging your interpreted Java program remotely, communication between the debugger and the program being debugged may be terminated prematurely by the JVM. If this happens, you will need to reload your program. To reload your program, select **File > Load Program** and enter the required information in to the **Load Program** dialog.

RELATED REFERENCES
irmtdbgc command

JAVA irmtdbgj command
idebug command

Starting the debugger on HP-UX or Solaris

The Distributed Debugger only supports remote debugging of interpreted Java programs on HP-UX and Solaris.

To start debugging a interpreted Java program on HP-UX or Solaris from the command line:

1. On the HP-UX or Solaris system, issue the `irmtdbgj` command at a command line prompt. If you issue this command without any parameters, you will be prompted for the required information in the Load Program dialog on the local system.
2. On the local system, issue the `idebug` command with the remote debug parameters at a command line prompt. You must specify the `-qhost` parameter.

The `idebug` command must be issued with `-qhost` option when debugging a remote program.

Tip: When debugging your interpreted Java program remotely, communication between the debugger and the program being debugged may be terminated prematurely by the JVM. If this happens, you will need to reload your program. To reload your program, select **File > Load Program** and enter the required information in to the **Load Program** dialog.

RELATED REFERENCES
irmtdbgj command
idebug command

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.

When you start the debug engine that will connect to this daemon, you must specify the same port number in the `-qport` option of the `irmtdbgc` command used to start the debug engine.

If you will be debugging an interpreted Java program, use the `irmtdbgj` command with `-qhost` and `-quiport` options. The port for the `irmtdbgj -quiport` option must be the same as the port for the `idebug -quiport` option.

RELATED TASKS
Starting the debugger for local debugging
Starting the debugger for remote debugging

RELATED REFERENCES
idebug command
irmtdbgc command

JAVA irmtdbgj command

Enabling and disabling debug on demand

WIN **Restriction:** Debug on demand is available only on Windows. It is only available for local debugging.

Debug on demand enables you to open a debugging session whenever an unhandled exception or other unrecoverable error occurs in your application.

To enable debug on demand, enter `idod idebug` at a command line prompt.

To disable debug on demand, enter `idod /u` at a command line prompt.

RELATED CONCEPTS

WIN Debug on demand

RELATED TASKS
Starting the debugger for local debugging

RELATED REFERENCES

WIN idod command

Debug on demand

WIN **Restriction:** Debug on demand is available only on Windows. It is only available for debugging programs locally.

Debug on demand enables you to open a debugging session whenever an unhandled exception or other unrecoverable error occurs in your program. The debugger starts and attaches to your program at the point of fault. This can save you time for two reasons: you do not have to recreate errors, and your program can run at full speed without interference from the debugger until the fault is encountered.

Debug on demand can be started for any application that fails while it is running, even if the application does not contain debug information. With debug on demand, you can even find and fix a problem in your application and let the application continue running.

RELATED TASKS
Enabling and disabling debug on demand

When to attach

Restriction: Attaching to a running process is only supported on AIX and Windows.

JAVA You can attach the debugger to an already running program or a running Java Virtual Machine (JVM) where an error or failure has occurred.

There are two main reasons for attaching the debugger to a process:
- 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.
- You are developing or maintaining a program that hangs sporadically, and you want to find out why it is hanging. In this situation, you can attach the debugger, and look for infinite loops or other problems that might be causing your program to hang.

WIN You can also use the debug on demand feature to invoke the debugger when an application running on your system throws an exception that is not handled. Debug on demand is not available when debugging interpreted Java programs.

AIX You can also use postmortem debugging to debug a core file containing information on the state of an application when it trapped. Postmortem debugging is available when debugging locally only. Postmortem debugging is not available when debugging interpreted Java programs.

RELATED TASKS
Attaching to a local running process
Attaching to a remote running process

JAVA Attaching to a running Java Virtual Machine

WIN Enabling and disabling debug on demand

Attaching to a local running process

JAVA **Important:** You can only attach to a running process when debugging a High Performance Compiled (HPC) Java program. For interpreted Java programs, you must attach to a running Java Virtual Machine (JVM). See the related topic below on how to attach to a running JVM.

You can attach the debugger to a running process either by using the Process List dialog or from a command line by using the `-a` option of the `idebug` command. See the related topic below on when to attach to a running process.

To attach the debugger to running process with the Process List dialog:
1. Select **File > Attach Process** to invoke the Process List dialog.
2. Select the process you want to attach from the **Select Process** list.
3. If the **Process Path** field is enabled, enter the full path name to the executable associated with the process you want to attach.
4. Click **Attach**.

To attach the debugger to a running process from a command line, enter the following command:

AIX idebug -a<process_id>

WIN `idebug -a<process_id> <full_path_to_executable>`

where `<process_id>` is a valid process id on your system and `<full_path_to_executable>` is the full path name to the executable associated with process id you want to attach.

Important: Do not attach to operating system processes or to the debugger's own process. Attaching to such processes can cause unpredictable results.

If you are currently debugging a process, that process is terminated when the new process is attached. When you exit the debugger, any attached process is also terminated.

You cannot restart a program that you have attached to.

RELATED CONCEPTS
When to attach

RELATED TASKS
Attaching to a remote running process
JAVA Attaching to a running Java Virtual Machine

RELATED REFERENCES
idebug command

Attaching to a remote running process

Restriction: Attaching to a running process is only supported on AIX and Windows.

JAVA **Important:** You can only attach to a running process when debugging a High Performance Compiled (HPC) Java program. For interpreted Java programs, you must attach to a running Java Virtual Machine (JVM). See the related topic below on how to attach to a running JVM.

You can attach the debugger to a process running on a remote system either by using the Process List dialog or from a command line by using the `-a` option of the `idebug` command. See the related topic below on when to attach to a running process.

To attach the debugger to a running process on a remote system with the Process List dialog:

1. On the remote system, start the debug engine using the `irmtdbgc` command. If you specify the `-qport` option, take note of it. You will need it later.
2. On the local system with the debugger user interface running, select **File > Attach Process** to invoke the Process List dialog.
3. In the Process List dialog, select **TCP/IP Connection**, then fill in the host name and port number for the remote system. If you specified the `-qport` option of the `irmtdbgc` command, the port number used in the `-qport` option must match the port number in Process List dialog.
4. Click **Refresh**.
5. Select the remote process you want to attach from the **Select Process** list.

6. If the **Process Path** field is enabled, enter the full path name to the executable associated with the process you want to attach.
7. Click **Attach**.

To attach the debugger to a running process from a command line:

1. On the remote system, start the debug engine using the `irmtdbgc` command. If you specify the `-qport` option, take note of it. You will need it later.
2. On the local system, enter the following command:

   AIX `idebug -qhost=<remote_host> [-qport=<host_port>] -a<process_id>`

   WIN `idebug -qhost=<remote_host> [-qport=<host_port>] -a<process_id> <full_path_to_executable>`

where `<remote_host>` is the the TCP/IP name or address of the remote system, `<full_path_to_executable>` is the full path name to the executable associated with process id you want to attach, and `<process_id>`is a valid process id on the remote system.

Important: Do not attach to operating system processes or to the debugger's own process. Attaching to such processes can cause unpredictable results.

If you are currently debugging a process, that process is terminated when the new process is attached. When you exit the debugger, any attached process is also terminated.

You cannot restart a program that you have attached to.

RELATED CONCEPTS
When to attach

RELATED TASKS
Attaching to a local running process
JAVA Attaching to a running Java Virtual Machine

RELATED REFERENCES
idebug command
irmtdbgc command

Attaching to a running Java Virtual Machine

JAVA **Important:** You can only attach to a Java Virtual Machine if you are debugging an interpreted Java program. For High Performance Compiled (HPC) Java programs, you must attach to a running process. See the related topic below on how to attach to a running process.

You can attach to an already running Java Virtual Machine (JVM) if you start your Java application with one of the following commands:

- on Java 1.1.x, use the `java_g -debug` command.
- for the Java 2 platform, use the following command:

  ```
  java -Xdebug -Djava.compiler=NONE
  -Xbootclasspath:<path\YourAppJarFile>;<path\dertrjrt.jar>;
  %JAVA_HOME%\lib\tools.jar;%JAVA_HOME%\jre\lib\rt.jar;%CLASSPATH%
  ```
  yourapp fdf

  where:

- `<path\YourAppJarFile>` is the fully qualified path to you client application JAR file.
- `<path\dertrjrt.jar>` is `/usr/idebug/lib/dertrjrt.jar` on AIX, `opt/IBMdebug/lib/dertrjrt.jar` on Solaris, and `x:\IBMDebug\lib\dertrjrt.jar` on Windows.
- *`yourapp`* is the fully qualified path to your program. For example, `C:\MyApps\MyAp`

When you start your application with one of these commands, an agent password is displayed. Take note of this password because it will be needed to attach to the running JVM.

Once your application is running and you have the agent password, you can connect to the JVM from the debugger user interface or from the command line.

To attach to a running JVM from the debugger user interface:

1. Select **Attach JVM** from the **File** menu. The Attach to Java Virtual Machine dialog appears.
2. Select the type of connection for the attach from the dialog.
3. Enter the required parameters for the type of connection.
4. Click **Attach** to attach to the JVM and dismiss the dialog.

To attach to a running JVM from the command line:

1. Start the debug engine daemon for the host JVM using the `irmtdbgj` command with the JVM attach parameters. If you are debugging remotely, remember to use the engine daemon parameters as well. For example:
   ```
   irmtdbgj -qhost=workstation_id -quiport=8001 -host=hostname
   -password=agent_password
   ```
   Where:
   - *`workstation_id`* is the TCP/IP name or TCP/IP address of the workstation running the debugger interface daemon.
   - *`hostname`* is the TCP/IP name or TCP/IP address of the machine running the JVM. If the host running the JVM and the machine running the debug engine are the same, use `localhost` as the *`hostname`*.
   - *`agent_password`* is the password printed when you start your Java program using the `java_g -debug` or `java -Xdebug` command.
2. Start the debugger interface in attach mode. Use a `process_id` of `0` for the `-a` parameter when attaching to a JVM. If you are debugging remotely, remember to use the remote debug parameters as well. For example:
   ```
   idebug -a0 -qhost=remotehost -qport=8001
   ```

RELATED CONCEPTS

When to attach

RELATED TASKS

Attaching to a local running process
Starting the debugger for local debugging
Starting the debugger for remote debugging

RELATED REFERENCES

idebug command

▶ JAVA irmtdbgj command

Debugging a Java applet

When debugging an applet, you must begin by debugging the **sun.applet.AppletViewer** class. After you have begun debugging this class, you can open the source for any class which is part of your applet and set breakpoints.

You can start to debug an applet by following these steps:

1. Issue the following command on your local system:

   ```
   idebug -qlang=java sun.applet.AppletViewer <applet_location>
   ```

   Where `<applet_location>` is one of the following:
   - a URL pointing to the applet you want to debug. The URL must begin with `http://` or `file:/` (only one slash for URLs beginning with `file`.)
   - the file name of the applet you want to debug only. The file must exist in the directory where you issued the `idebug` command.
2. Click **OK**.
3. From the main menubar, select **Source > Open New Source**.
4. Enter the name of the applet class you are debugging.
5. Click **OK**.
6. Set your breakpoint and run.

We recommend that you set a breakpoint on the `init()` or `start()` methods since these are the first methods that are called by the applet viewer.

Once you have set this breakpoint you can debug your applet as you would debug a Java application.

RELATED REFERENCES
idebug command

Registering a class or package

▶ JAVA **Restriction:** This is supported for Java only.

You can only debug Java classes that have been registered with the debugger or classes that are part of packages that have been registered with the debugger. Registered classes and packages are those that appear in the Packages pane.

To register a class or package with the debugger:

1. Select **Source > Open New Source**.
2. Check **All packages**.
3. Enter the fully-qualified class name for the class you want to register in the **Class** field of the Open New Source dialog box.
4. Click **OK** to load the class and its associated package and dismiss the dialog box.

If you want to register more than one class with the debugger, click **Open** to register the class you just entered. Then register your next class. When you have registered all your classes, click **OK**.

If the class you entered is part of a package, the entire package is registered with the debugger as well as the class you entered.

Chapter 4. Working with breakpoints

Breakpoints

Breakpoints are temporary markers you place in your executable program to tell the Distributed 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 check the contents of variables, registers, storage, and the stack. You can then step over (execute) the statement to see how the problem arises or you can choose to skip the execution of the statement in question.

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.
- C C++ **Function breakpoints** are triggered when the function they apply to is entered.
- JAVA **Method breakpoints** are triggered when the method they apply to is reached.
- **Storage change breakpoints** are triggered when storage at a specified address is changed.

  AIX Storage change breakpoints are not available when debugging programs running on AIX.

  JAVA Storage change breakpoints are not available when debugging interpreted Java programs.
- **Load occurrence** breakpoints are triggered when a DLL is loaded.

  JAVA Load occurrence breakpoints are not supported when debugging interpreted Java.
- AIX WIN **Address breakpoints** are triggered before the disassembly instruction at a particular address is executed.

  JAVA Address breakpoints are not available when debugging interpreted Java.

You can set conditions on line breakpoints. When you run the program, execution stops at the breakpoint before the statement is executed if the breakpoint condition is met.

RELATED TASKS
Setting a line breakpoint

C C++ Setting a function breakpoint

JAVA Setting a method breakpoint
Setting a storage change breakpoint
Setting a load occurrence breakpoint
Setting an address breakpoint
Setting a conditional breakpoint

Setting breakpoints

Setting a line breakpoint

You can set line breakpoints from the Source pane, the Source menu and the Breakpoints menu.

To set a line breakpoint in the Source pane:

1. On AIX and Windows ensure the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View**.
2. Make sure the appropriate line is visible in the Source pane by using the scroll bar or cursor keys to locate the line.
3. 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.

To set a line breakpoint from the Source menu:

1. Select **Source > Set Line Breakpoint** from the menu bar.
2. Enter the name of the module or routine in which you want to set a breakpoint in the **Executable** entry field in the Line Breakpoint dialog. If this module or routine is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. Choose or enter the object, class or source file that is associated with the module or routine specified in the **Executable** entry field and contains the line where the breakpoint is to be set from the **Source** pulldown list.
4. Choose the source file containing the code for the object or class file from the **Include File** pulldown list. (This step is optional if you have not selected to defer the breakpoint.)
5. Enter the line number within the source file where you want to place a breakpoint in the **Line** entry field.
6. If the module or routine you entered in the **Executable** entry field is not currently loaded, click on the **Defer breakpoint** check box.

   JAVA This check box is not available when debugging interpreted Java.
7. Set any optional parameters that you want for the breakpoint.
8. Click **OK** to set the breakpoint and dismiss the Line Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Line Breakpoint dialog.

To set a line breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Line** from the menu bar.
2. Enter the name of the module or routine in which you want to set a breakpoint in the **Executable** entry field in the Line Breakpoint dialog. If this module or routine is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. Choose or enter the object, class or source file that is associated with the module or routine specified in the **Executable** entry field and contains the line where the breakpoint is to be set from the **Source** pulldown list.
4. Choose the source file containing the code for the object or class file from the **Include File** pulldown list. (This step is optional if you have not selected to defer the breakpoint.)

5. Enter the line number within the source file where you want to place a breakpoint in the **Line** entry field
6. If the module or routine you entered in the **Executable** entry field is not currently loaded, click on the **Defer breakpoint** check box.

   JAVA This check box is not available when debugging interpreted Java.
7. Set any optional parameters that you want for the breakpoint.
8. Click **OK** to set the breakpoint and dismiss the Line Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Line Breakpoint dialog.

JAVA When debugging your interpreted Java program, the debugger will ignore breakpoints set on static initializers, static blocks and `try` statements.

RELATED CONCEPTS
Breakpoints

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

Setting a function breakpoint

You can set function breakpoints from the Modules pane, the Source menu and the Breakpoints menu.

To set a function breakpoint from the Modules pane:
1. Expand the list in the Modules pane until you see the function you want.
2. Right-click on that function and select **Set Function Breakpoint** from the pop-up menu.

To set a function breakpoint from the Source menu:
1. Select **Source > Set Function Breakpoint** from the menu bar.
2. Enter the name of the executable which contains the function where you want to set a breakpoint in the **Executable** entry field in the Function Breakpoint dialog. If this executable is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. Choose or enter the object, class or source file for the executable specified in the **Executable** entry field from the **Source** pulldown list.
4. Enter the name of the function where the breakpoint is to be set in the **Function** entry field in the Function Breakpoint dialog. If this function is loaded, you can select it from the pulldown list in the **Function** entry field.
5. If the executable or DLL containing the function you want to debug is not currently loaded, click on the **Defer breakpoint** check box.
6. Set any optional parameters that you want for the breakpoint.
7. Click **OK** to set the breakpoint and dismiss the Function Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Function Breakpoint dialog.

To set a function breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Function** from the menu bar.
2. Enter the name of the executable which contains the function where you want to set a breakpoint in the **Executable** entry field in the Function Breakpoint dialog. If this executable is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. Choose or enter the object, class or source file for the executable specified in the **Executable** entry field from the **Source** pulldown list.
4. Enter the name of the function where the breakpoint is to be set in the **Function** entry field in the Function Breakpoint dialog. If this function is loaded, you can select it from the pulldown list in the **Function** entry field.
5. If the module or routine you entered in the **Executable** entry field is not currently loaded, click on the **Defer breakpoint** check box.
6. Set any optional parameters that you want for the breakpoint.
7. Click **OK** to set the breakpoint and dismiss the Function Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Function Breakpoint dialog.

RELATED CONCEPTS
Breakpoints

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

Setting a method breakpoint

JAVA **Restriction:** This is supported for Java only.

You can set method breakpoints from the Packages pane, the Source menu and from the Breakpoints menu.

To set a method breakpoint from the Packages pane:

1. Expand the list in the Packages pane until you see the method you want to set an method breakpoint on.
2. Right-click on that method and select **Set Method breakpoint** from the pop-up menu.

To set a method breakpoint from the Source menu:

1. Select **Source > Set Method Breakpoint** from the menu bar.
2. Enter the name of the method you want to debug in the **Method** entry field in the Method Breakpoint dialog. If this method is loaded, you can select it from the pulldown list in the **Method** entry field.
3. If the Method you chose in the Method pulldown list is not currently loaded, click on the **Defer breakpoint** check box.
4. Set any optional parameters that you want for the breakpoint.
5. Click **OK** to set the breakpoint and dismiss the Method Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Method Breakpoint dialog.

To set a method breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Method** from the menu bar.
2. Enter the name of the executable you want to debug in the **Executable** entry field in the Method Breakpoint dialog. If this executable to be debugged is loaded, you can select it from the pulldown list in the **Executable** entry field.
3. Choose or enter the object, class or source file that is associated with the module or routine specified in the **Executable** entry field and contains the line where the breakpoint is to be set from the **Source** pulldown list.
4. Choose the source file containing the code for the object or class file from the **Include File** pulldown list.
5. Enter the name of the method you want to debug in the **Method** entry field in the Method Breakpoint dialog. If this method is loaded, you can select it from the pulldown list in the **Method** entry field.
6. Set any optional parameters that you want for the breakpoint.
7. Click **OK** to set the breakpoint and dismiss the Method Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Method Breakpoint dialog.

RELATED CONCEPTS
Breakpoints

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

Setting an address breakpoint

Restriction: This is supported for AIX and Windows only.

JAVA Address breakpoints are not available when debugging interpreted Java programs. They are available when debugging High Performance Compiled (HPC) Java programs.

You can set an address breakpoint from the Source pane, the Source menu, and from the Breakpoints menu.

To set an address breakpoint in the Source pane:

1. Ensure the Source pane is set to a disassembly or mixed view. To set the Source pane to a disassembly view, select **Source > Disassembly View**. To set the Source pane to a mixed view, select **Source > Mixed View**.
2. Make sure the appropriate line is visible in the pane by using the scroll bar or cursor keys to locate the line.
3. Double-click on the line number in the prefix area of the line.

To set an address breakpoint from the Source menu:

1. Select **Source > Set Address Breakpoint** from the menu bar.
2. Enter either the address where you want to set the breakpoint or an expression that evaluates to an address. The address must be entered in hexadecimal notation.

3. Set any optional parameters that you want for the breakpoint.
4. Click **OK** to set the breakpoint and dismiss the Address Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Address Breakpoint dialog.

To set an address breakpoint from the Breakpoints menu:
1. Select **Breakpoints > Set Address** from the menu bar.
2. Enter either the address where you want to set the breakpoint or an expression that evaluates to an address. The address must be entered in hexadecimal notation.
3. Set any optional parameters that you want for the breakpoint.
4. Click **OK** to set the breakpoint and dismiss the Address Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Address Breakpoint dialog.

RELATED CONCEPTS
Breakpoints

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

Setting a storage change breakpoint

JAVA Storage change breakpoints are not available when debugging interpreted Java programs. They are available when debugging High Performance Compiled (HPC) Java programs.

Storage change breakpoints halt execution of your program whenever storage at a specific address is changed. For example, if a byte being watched contains X’40’ and the program writes X’40’ to that byte, the storage change breakpoint is not triggered. If the program writes X’41’, the storage change breakpoint is triggered.

To set a storage change breakpoint from the Breakpoints menu:
1. Select **Breakpoints > Set Storage Change** from the menu bar.
2. Enter an address or expression that evaluates to an address in the **Address or Expression** field.

   C C++ **Tip:** You can enter the address of a variable by specifying the variable name preceded by an ampersand (&).
3. Specify the number of bytes to be monitored in the **Bytes to Monitor** field.
4. Set any optional parameters that you want for the breakpoint.
5. Click **OK** to set the breakpoint and dismiss the Storage Change Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Storage Change Breakpoint dialog.

Caution: If you set a storage change breakpoint for any address that is on the call stack, be sure to remove the breakpoint before leaving the routine associated with it. Otherwise, when you return from the routine, the routine’s stack frame will be removed from the stack, but the breakpoint will still be active. Any other routine that gets loaded on the stack will then contain the breakpoint.

RELATED CONCEPTS
Breakpoints

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

Setting a load occurrence breakpoint

JAVA Load Occurrence Breakpoints are not available when debugging interpreted Java programs. They are available when debugging High Performance Compiled (HPC) Java programs.

Load occurrence breakpoints halt execution of your program when the DLL or dynamically loaded module specified is loaded into memory. You can set load occurrence breakpoints from the Breakpoints menu.

To set a load occurrence breakpoint from the Breakpoints menu:

1. Select **Breakpoints > Set Load Occurrence** from the menu bar.
2. Enter the name of the DLL or dynamically loaded module to set the breakpoint for.
3. Set any optional parameters that you want for the breakpoint.
4. Click **OK** to set the breakpoint and dismiss the Load Occurrence Breakpoint dialog. Alternatively, use the **Set** button to set the breakpoint without dismissing the Load Occurrence Breakpoint dialog.

RELATED CONCEPTS
Breakpoints

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

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. On AIX and Windows ensure the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View**.
2. Use the Breakpoints menu to select the type of breakpoint that you want to set.

3. On the Breakpoint dialog complete any or all optional parameters that you want as conditions for your breakpoint.
4. Click OK to set the conditional breakpoint and dismiss the dialog.

RELATED CONCEPTS
Breakpoints

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

Setting a deferred breakpoint

A deferred breakpoint is a breakpoint set in a DLL, dynamically called routine or executable that is not currently loaded. You can defer the following types of breakpoints:

- line breakpoints
- function breakpoints
- method breakpoints

To set a deferred breakpoint, click on the **Defer breakpoint** check box when setting one of the above types of breakpoints.

RELATED CONCEPTS
Breakpoints

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

Setting multiple breakpoints

You can set several breakpoints with the same optional parameters from any of the breakpoint dialogs.

To set multiple occurrences of a type of breakpoint:

1. Select the type of breakpoints you want to set from either the **Source** menu or the **Breakpoints** menu.
2. From the Breakpoint dialog, enter the required information for the first breakpoint. Change any fields in the **Optional Parameters** section of the dialog, 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, new function , new method , or new address) and click on **Set**.
5. After you have set the last breakpoint, click on **Cancel** to dismiss the dialog.

RELATED CONCEPTS
Breakpoints

RELATED TASKS
Modifying breakpoint properties
Enabling and disabling breakpoints
Setting a conditional breakpoint
Deleting a breakpoint
Viewing breakpoints

Viewing set breakpoints

A list of breakpoints you have set is kept in the **Breakpoints** pane for the process you are debugging. This list is originally collapsed and can be expanded to show your installed breakpoints. The 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. Click on the **Breakpoints** tab for the process or program you are debugging.
2. Expand or collapse the list of breakpoints to display the breakpoints you want to see.

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

RELATED CONCEPTS
Breakpoints

RELATED TASKS
Setting multiple breakpoints
Modifying breakpoint properties
Enabling and disabling breakpoints
Deleting a breakpoint

Modifying breakpoint properties

You can change the following properties of a breakpoint:
- Which threads the breakpoint applies to.
- How often the debugger should skip the breakpoint (the frequency).
- Whether to stop on the breakpoint only when a given expression is true. Expressions can only be applied to the following breakpoints:
  - line breakpoints
  - function breakpoints
  - method breakpoints
  - address breakpoints
- Whether to defer the breakpoint. Only the following breakpoints can be deferred:
  - line breakpoints
  - function breakpoints
  - method breakpoints

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 breakpoint's properties:

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 modify.
3. Right-click on the breakpoint you want to modify.
4. Select **Modify Breakpoint** from the pop-up menu. A Breakpoint dialog corresponding to the breakpoint type appears displaying the current settings for the breakpoint.
5. Change the breakpoint's properties in the Breakpoint dialog.

RELATED CONCEPTS
Breakpoints

RELATED TASKS
Setting multiple breakpoints
Enabling and disabling breakpoints
Deleting a breakpoint
Viewing breakpoints

Enabling and disabling breakpoints

You can disable a breakpoint so that it does not stop execution, and then later enable it again. 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 breakpoints from the Breakpoints pane. Also, you can enable or disable breakpoints from the Source pane.

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.
3. Right-click on the breakpoint you want to enable or disable.
4. Select **Enable Breakpoint** or **Disable Breakpoint** from the pop-up menu.

To enable or disable a breakpoint from the Source pane:

1. Ensure the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View**.
2. Scroll to the line which contains the breakpoint you want to enable or disable.
3. Right-click on the line which contains the breakpoint you want to enable or disable.
4. Select **Enable Breakpoint** or **Disable Breakpoint** from the pop-up menu.

To enable all breakpoints, select **Breakpoints > Enable All Breakpoints** from the menu bar.

To disable all breakpoints, select **Breakpoints > Disable All Breakpoints** from the menu bar.

RELATED TASKS
Deleting a breakpoint

Deleting a breakpoint

You can delete single breakpoints from the Source pane and the Breakpoints pane. All breakpoints can be deleted at once 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 execution, disable the breakpoint instead. For information on disabling breakpoints, see the related topic below.

To delete a single breakpoint in the Source pane:

1. Scroll to the line which contains the breakpoint you want to delete.
2. Do one of the following to delete the breakpoint:
   - Double-click on the line number in the prefix area of the line to delete the breakpoint.
   - Right-click on the breakpoint and select **Delete Breakpoint** from the pop-up menu.

To delete a single breakpoint 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 breakpoints by clicking on the plus icons (⊞) until you see the breakpoint you want to delete.
3. Right-click on the breakpoint you want to delete.
4. Select **Delete Breakpoint** from the pop-up menu.

To delete all breakpoints, select **Breakpoints > Delete All Breakpoints** from the menu bar.

If you want to temporarily prevent all breakpoints from stopping execution, disable them instead by selecting **Breakpoints > Disable All Breakpoints**.

RELATED TASKS
Enabling and disabling breakpoints

Chapter 5. 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 select **Debug > Run**, the program will run until the end of the program is reached, an active breakpoint is hit, or an exception occurs if exception filtering is set to a level other than NONE.

To run a program until an active breakpoint is hit, do one of the following:

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

If you select **Run to Location**, 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.

To run a program to a specific line number:
1. Ensure the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View** from the menu bar.
2. 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.
3. Run the program to the line by doing one of the following:
   - Right-click on the line to bring up 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 press F10.

RELATED CONCEPTS
Breakpoints

RELATED TASKS
Stepping through a program
Setting multiple breakpoints
Modifying breakpoint properties
Enabling and disabling breakpoints
Deleting a breakpoint
Viewing breakpoints

Exception Handling

The Distributed Debugger allows you to investigate exceptions that occur while you are debugging your program. Only types of exception or the level of exception you want the debugger to recognize will be handled.

You can choose the types of exception or the level of exception you want the debugger to recognize in the Exception Filter Preferences Setting field in Applications Preferences dialog box. The types or exception or level of exception you can select varies with the platform the program you are debugging run on. For example, the exceptions the debugger can handle for a C++ program running on Windows NT® are different from the exceptions the debugger can handle for a C++ program running on AIX.

When the debugger encounters an exception that matches one of the exceptions that are specified in the Exception Filter Preferences Settings dialog box, a dialog box opens to warn you an exception occurred. Also, the line where the exception occurred remains highlighted in the Source pane.

After a program exception is encountered and the Application Exception Occurred dialog box is closed, the following actions are available:

Step exception
Step exception causes the debugger to step into the first registered exception handler (tracked by the operating system). Execution stops at the first executable line of code in the exception handler. If your application does not have a registered condition handler, the condition remains “unhandled,” the application may be terminated.

JAVA When debugging interpreted Java programs, exceptions are handled through a try/catch/finally mechanism. Selecting Step Exception causes the debugger to step into the first catch clause that handles the exception that was thrown. Execution stops at the first executable line of code in the catch block. If the exception is not caught by your application, the exception remains “unhandled” and the application may be terminated.

Run exception
Run exception causes the debugger to run the exception handler that is registered to handle the type of exception encountered. If your application does not have a registered condition handler, the condition remains “unhandled,” the application may be terminated.

JAVA When debugging interpreted Java, selecting this action causes the debugger to run the catch block that handles the type of exception encountered. If your application does not have a registered condition handler, the condition remains “unhandled,” the application may be terminated.

Examine/Retry exception
Examine/Retry exception allows you to investigate the cause of the exception and, if desired, retry program execution at the statement that triggered the exception. The debugger begins at this statement and attempts to continue.

Stepping through a program

You can use **step commands** to step through your program a single statement at a time. The statements can be source code or disassembly instructions. 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 Into command, do one of the following:

- Click the step into button ( ) on the toolbar.
- Select **Debug > Step Into** from the menu bar.
- Press F11.

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.

To execute a Step Return command, do one of the following:

- Click the set return button ( ) on the toolbar.
- Select **Debug > Step Return** from the menu bar.
- Press Shift+F11.

JAVA When debugging your interpreted Java program, stepping behavior may be erratic when stepping into constructors, or when stepping into or over `SystemLoad` library functions.

JAVA **Tip:** If you step into a class that has not been registered with the debugger, you may receive a **Cannot find source for null** message. If this happens, issue a step return command to continue debugging. To avoid this problem, register the appropriate source file for the class or package containing that class) with the debugger before you start debugging.

RELATED TASKS

Running a program

JAVA Registering a class or package

RELATED REFERENCES

Step commands

Step commands

You can use **step commands** to step through your program a single line or on AIX or Windows disassembly instruction at a time.

The following types of step commands are available:

Step Command	Button	Shortcut	Description
Step Over		F10	Executes the current line, without stopping in any functions or routines called within the line.
Step Into		F11	Executes the current line. If the current line contains a call to a function or routine, execution stops in the first line or disassembly instruction of the called function or routine. If the called function or routine was not compiled with debug information, the function or routine is shown in a disassembly view. Since disassembly views are not available for Java, you will see a "No Source Available" message if the called function does not exist in a path in the CLASSPATH environment variable.
Step Debug		F7	Executes the current line. Execution stops at the next line encountered for which debug information is available. This could be in the current function or routine, in the called function or routine, or in a function or routine called within the called function or routine.
Step Return		Shift+F11	Executes from the current execution point up to the line immediately following the line that called this function or routine. If you issue a Step Return command from the main entry point (in C++, the main() program), the program runs to completion.

Execution of your program may stop earlier than indicated in the step command descriptions, if the Distributed Debugger encounters a breakpoint or an exception occurs.

You can use combinations of step commands to step through multiple calls on a single line.

RELATED TASKS
Stepping through a program

Skipping over sections of a program

You can skip over sections of code to avoid executing certain statements or to move to a position that certain statements can be executed again.

To skip over a section of code:
1. Ensure the Source pane is set to source view. To set the Source pane to source view, select **Source > Source View** from the menu bar.
2. Scroll to the line where there are statements that you want to avoid executing or you want to execute again.
3. Jump to the line by doing one of the following:
   - Right-click on the line and select **Jump to Location** from the pop-up menu.
   - Click on the line to select it, then select **Debug > Jump to Location** from the menu bar.

Using Jump to Location can cause unpredictable results if you jump outside the current method, jump over code that has side-effects (for example, calls to methods whose results are assigned to variables, or methods that change the contents of variables passed by reference), or jump into the middle of a block such as a **for** loop.

RELATED TASKS
Running a program

Halting execution of a program

Halting a program stops the execution of the program without terminating the execution of the program. It allows you to pause and examine the program's internal state and then continue execution without restarting the program.

To halt execution of a program that is currently running in the debugger, do one of the following:

- Click on .
- Select **Debug > Halt** from the menu bar.

You may find that execution halts in a function other than the one you are debugging (for example, a system library routine). To run to the end of that routine and stop in your own code, do one of the following:

- Issue a Step Return command.
- If the previous technique results in the debugger displaying the message "Cannot determine return address", issue the Step Debug command until execution returns to your code
- If you know what line in your program will be the next to execute after the current function returns, go to the source pane containing that line, set a breakpoint on it, and issue the Run command.

JAVA **Important:** The debugger cannot halt an applet or application that has all of its threads blocked.

RELATED CONCEPTS
Step commands

RELATED TASKS
Running a program
Stepping through a program
Setting a line breakpoint

Restarting a program

To start debugging your program from the beginning if your program is stopped, click on in the toolbar or select **Debug > Restart** from the menu bar.

To start debugging your program again from the beginning, if your program is not stopped:

1. Issue a Halt command by clicking on in the toolbar or selecting **Debug > Halt** from the menu bar, if the program is currently executing within the debugger.
2. Set a breakpoint at the location you want to run to, if it is not the beginning of your program and you have not already set a breakpoint there.
3. Click on in the toolbar or select **Debug > Restart** from the menu bar.

If the previous run of your program produced side effects such as the creation of an output file and the program logic will be changed by the existence of such files from a previous debug session, you may want to erase these files before restarting.

RELATED TASKS

Halting execution of a program
Running a program

Chapter 6. Inspecting data

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.

Local variables that are in scope can also be monitored in the Locals pane. By default, all local variables in scope are added to the Locals pane.

To add a variable or expression to the Monitors pane from the Source pane:

1. Ensure the Source pane is set to source view. To set the Source pane to a source view, select **Source > Source View** from the menu bar.
2. Highlight the variable or expression you want to monitor.
3. 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. Select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, enter the variable or expression you want to monitor.
3. Select the **Program monitor** radio button.
4. Click **OK** to add the variable or expression to the monitor and dismiss the dialog.

the Monitors pane from the Monitors menu:

1. Select **Monitors > Monitor Expression** from the menu bar.
2. In the dialog, 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.

RELATED REFERENCES
C/C++ supported data types
C/C++ supported expression operands
C/C++ supported expression operators
Interpreted Java expressions supported

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. Expand the thread in the Locals pane where the local variable you want to view appears.
2. If necessary, use the scroll bars or PageUp and PageDown keys to scroll the pane until the variable is visible.
3. If your variable is a class, struct 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 a representation from the **Monitor Representation** menu.

variable or expression you have already added to the Monitors pane:

1. Use the scroll bars or PageUp and PageDown keys to scroll the pane until the variable is visible.
2. If your variable is a class, struct or array, it can be expanded to show its individual elements.
3. If desired, change the representation of the variable: right-click on the variable and select a representation from the **Monitor Representation** menu.

message displays in the Monitors pane instead of a value.

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

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

RELATED TASKS
Enabling hover help for variables
Adding a variable or expression to the Monitors pane
Viewing a location in storage

Changing the contents of a variable

Note: The contents of variables cannot be changed when debugging an interpreted Java program.

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

1. Expand the monitor containing the variable whose value you want to modify.
2. If your variable is a class, struct or array, expand it to show its individual elements.
3. Scroll down to the variable you want to change and 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.
4. Enter a value that is valid for the current representation of that variable or variable element.
5. Press Enter to submit the change. The debug engine checks for a valid value.

RELATED TASKS
Adding a variable or expression to the Monitors pane

Inspecting registers

Viewing the contents of a register

JAVA **Note:** Register information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

You can view the contents of a register from the Registers pane, the Monitors pane if you have added the register there, or a Storage Monitor pane if you have added the register there.

To view the contents of a register in the Registers pane:
1. Expand the thread for which you want to view the registers.
2. Expand the register category that contains the register you want to view.
3. If desired, use the scroll bars or PageUp and PageDown keys to scroll the pane until the register is visible.

To view the contents of a register you have already added to the Monitors pane:
1. If necessary, use the scroll bars or PageUp and PageDown keys to scroll the Monitors pane until the register is visible.
2. If desired, change the representation of the register: right-click on the register and select a representation from the **Monitor Representation** menu.

To view the contents of a register you have already added to a Storage pane:
1. If necessary, use the scroll bars or PageUp and PageDown keys to scroll the Storage pane until the register is visible.
2. If desired, change the representation of the register: right-click on the register and select a representation from the **Monitor Representation** menu.

RELATED TASKS
Adding a new Storage Monitor pane for an expression or register
Adding a register to the Monitors pane

Viewing the contents of a floating-point register

OS/2 WIN **Restriction:** This section applies to programs running on Intel processors only (OS/2 and Windows).

JAVA Register information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

If you are debugging an Intel-based application and you step over a source line containing floating-point arithmetic, you may find that the values of floating-point registers in the **Registers** pane are not displayed. Instead, "Not used" appears beside each register. In fact, one or more of these registers is being used during execution of the source line, but once you have stepped over the source line, the register's contents have been written to a variable and the register is no longer in use. If you want to step over a floating-point statement and see a floating-point register's value before it is written to the variable, do the following:
1. Change from source view to mixed view (select **Source > Mixed View**).

2. Locate the source line containing the floating-point instructions. Look for a disassembly instruction between this source line and the next that contains a floating-point store instruction (for example `FSTP`), and place a breakpoint on that line.
3. Change back to source view.
4. Now when you step over the source line containing the floating-point arithmetic, you must issue two Step Over commands for the line instead of one (because the first Step Over command stops at the breakpoint you set in the mixed view). After the second Step Over command, you should see the value of the floating point register as it was before it was stored.

RELATED TASKS
Changing the contents of a register

Changing the contents of a register

JAVA **Note:** Register information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

To change the contents of a register in the Registers pane, Monitors pane or Storage Monitor pane:

1. In the Registers pane, Monitors pane, or Storage Monitor pane expand the entry which contains the register whose value you to want.
2. Scroll to the register you want to change and do one of the following:
   - Double-click on the register.
   - Select **Edit** from the pop-up menu.
3. Enter a value that is valid for the current representation of that register.
4. Press **Enter** to submit the change. The Distributed Debugger checks for a valid value.

RELATED TASKS
Viewing the contents of a register
Adding a register to the Monitors pane

Adding a register to the Monitors pane

JAVA **Note:** Register information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

You can add a register to the Monitors pane if you want to monitor only a few registers during the execution of your program. Registers can also be monitored in the Registers pane and Storage pane. To monitor all registers during program execution, use the Registers pane.

To add a register to the Monitors pane:

1. Click on the **Monitors** tab and do one of the following:
   - Select **Monitors > Monitor Expression** from the menu bar.
   - Press Shift+F9.
2. In the dialog, enter the name of the register you want to monitor. Check the Registers pane to see a valid Registers name.

3. Select **Program Monitor**.
4. Click **OK** to add the register to the Expressions monitor and dismiss the dialog.

To add multiple registers to the Monitors pane:
1. Click on the **Monitors** tab and do one of the following:
   - Select **Monitors > Monitor Expression** from the menu bar.
   - Press Shift+F9.
2. In the dialog, enter the register you want to monitor.
3. Click **Monitor** to add the register to the monitor.
4. Repeat steps 2 and 3 until you have added all the registers you want to monitor.
5. Click **Cancel** to dismiss the dialog.

RELATED TASKS
Viewing the contents of a register
Adding a new Storage Monitor pane for an expression or register

Inspecting storage

Viewing a location in storage

JAVA **Restriction:** Storage information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

You can view the contents of storage from the Storage pane or from a new Storage Monitor pane that you have created.

To view the contents of storage from the Storage pane:
1. If desired, change the representation of the storage contents in the Storage pane.
2. If necessary, use the scroll bar in the Storage pane to view storage locations above or below the starting address of the Storage pane.
3. You can jump directly to an address in the Storage pane by doing the following:
   - Double-click on any address field in the Storage pane.
   - Enter the address you want to view. This address can be an expression, for example &x.
   - Press Enter. The storage contents now shown in the Storage pane are centered around the address you just entered.

To view the contents of storage from a Storage Monitor pane that you have created:
1. If desired, change the representation of the storage contents in the Storage Monitor pane.
2. If necessary, use the scroll bar in the Storage Monitor pane to view storage locations above or below the starting address of the Storage Monitor pane.
3. Use the **Go to Address** button to return to the starting address of the Storage Monitor pane.

▶ C ▶ C++ To view the *contents* of a C or C++ variable, such as an integer, in a Storage monitor precede the variable with an ampersand (&), or select a pointer that points to that variable. For example, given the following C or C++ source code:

```
int i=10;
int* p=&i;
```

You can monitor the storage for the variable `i` by entering either `&i` or `p` in the Monitor expression dialog, then selecting the **Storage monitor** radio button in that dialog.

RELATED TASKS
Changing the representation of storage contents
Changing the contents of a storage location
Creating a new Storage monitor for an expression or register

Changing the representation of storage contents

▶ JAVA **Note:** Storage information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

For each Storage monitor or Storage Monitor pane you have, you can change the representation of the storage and the number of columns shown in each pane.

These settings affect only the Storage monitor or Storage Monitor pane you are viewing, so you can have multiple Storage Monitor panes with different settings.

- Select the representation of storage for the Storage pane or Storage Monitor pane you are viewing from the **Content style** pulldown list. The **Content Style** pulldown list is at the bottom of the pane.
- Select the number of columns shown in a Storage pane or Storage Monitor pane from the **Columns Per Line** pulldown list. The **Columns Per Line** pulldown list is at the bottom of the pane.

Changing the contents of a storage location

▶ JAVA **Note:** Storage information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

To change the contents of a storage location in a Storage pane or Storage Monitor pane:

1. Select the Storage pane or Storage Monitor pane where you want to make the change.
2. Scroll down to the storage location you want to change.
3. Double-click on the value you want to change.
4. Enter a valid value for that storage location.
5. Press **Enter** to submit the change. The Distributed Debugger checks for a valid value.

Adding a new Storage Monitor pane for an expression or register

▶ JAVA **Restriction:** Storage or register information is not available when debugging interpreted Java programs. It is available when debugging High Performance Compiled (HPC) Java programs.

Registers can be monitored in the Registers pane and the Monitors pane. To monitor all registers during program execution, use the Registers pane.

You may want to add a new Storage Monitor pane for an expression or register if you want to monitor specific locations in storage or only a few registers during the execution. To monitor all locations in storage during program execution, use the Storage pane.

Warning: If there is a variable in scope which has the same name as the register that you are trying to use, the variable will be used first.

To add a new Storage Monitor pane for an expression or register from the Registers pane:

1. Highlight the expression or register you want to add a new Storage Monitor pane.
2. Right-click on the highlighted expression or register and select **Add to Storage Monitor** from the pop-up menu. A new Storage Monitor pane will appear with the expression or register appearing in the monitor's tab.

To add a new Storage Monitor pane for an expression or register from the Monitors pane:

1. Click on the **Monitors** tab and do one of the following:
   - Select **Monitors > Monitor Expression** from the menu bar.
   - Press Shift+F9.
2. In the dialog, enter the expression or register that you want to monitor.
3. Select the **Storage Monitor** radio button.
4. Click **OK** to add the new Storage Monitor pane.
5. A new Storage Monitor pane will appear with the expression or register appearing in the monitor's tab.

To add multiple new Storage Monitor panes from the Monitors pane:

1. Click on the **Monitors** tab and do one of the following:
   - Select **Monitors > Monitor Expression** from the menu bar.
   - Press Shift+F9.
2. In the dialog, enter the first expression or the name of the register that you want to monitor.
3. Select the **Storage Monitor** radio button.
4. Click **Monitor** to add the new Storage Monitor pane for the expression or register entered.
5. Repeat steps 2 to 4 until you have added all the storage locations or registers that you want to monitor.
6. Click **Cancel** to dismiss the dialog.

Tip: Check the Registers pane to see a valid Registers name.

RELATED TASKS
Changing the representation of storage contents
Changing the contents of a storage location
Viewing a location in storage

Heap errors

Restriction: This information applies to debugging C or C++ programs on AIX and Windows only.

Heap errors can occur when your code inadvertently overwrites control information that the memory management functions use to control heap usage. Each block of allocated storage within a heap consists of a data area, which starts at the address returned by the allocating function, as well as a control area adjacent to the data area, which is needed by the memory management functions to free the storage properly when you deallocate the storage. If you overwrite a control structure in the heap (for example, by writing to elements outside the allocated bounds of an array, or by copying a string into too small a block of allocated storage), the control information is corrupted and may cause incorrect program behavior even if the data areas of other allocated blocks are not overwritten.

You should consider the following points when you are trying to locate heap errors:

Finding heap errors outside the debugger

To detect heap errors, you can compile your program to use the heap-checking versions of memory management functions. When you run a program compiled with this option, each call to a memory management function causes a heap check to be performed on the default heap. This heap check involves checking the control structures for each allocated block of storage within the heap, and ensuring that none was overwritten. If an error is encountered, the program terminates and information is written to standard error including the address where heap corruption occurred, the source file and line number at which a valid heap state was last detected, and the source file and line number at which the memory error was detected.

Heap checking for default and other heaps

Heap checking is only enabled for the default heap used by each executable. If the debug versions of the memory management functions do not report heap corruption and you still suspect a problem, you may be using additional heaps and corrupting them. You can debug usage of nondefault heaps by adding calls to the **_uheapchk** C Library function to your source code. See your compiler documentation for more information.

Pinpointing heap errors within the debugger

You can pinpoint the cause of a heap error from within the debugger, provided the heap causing the error is known to be the default heap, by continually narrowing down the gap between the last line at which the heap was valid, and the first line at which corruption occurred. From within the Source pane, use a combination of run commands, step commands, line and function breakpoints, and the **Check heap when stopping** setting on the **Run** menu, to narrow the scope of your search.

Check heap when stopping may expose other coding errors

For semantically incorrect programs, **Check heap when stopping** is intrusive in that it may cause different results where a program is incorrectly accessing data on the stack. This is because **Check heap when stopping** causes the process and thread being debugged to call a heap check function each time execution stops, and this heap check function affects the safe area of the stack by overwriting part of that area with its stack frame. For example, if a called function returns the address of a local variable, that local variable's contents will be accessible from the calling function, and will not change, as long as the stack frame used by the called function is not overwritten by a subsequent call. However, if you issue a Step return command from the called function while **Check heap when stopping** is enabled, the heap checking function is called immediately on return from the called function, and the storage pointed to by the returned pointer may overwritten by the stack frame of the heap checking function.

Check heap when stopping affects performance

Heap checking within the debugger has a high overhead cost for step commands, because the heap is checked after each step. If you are stepping through large sections of code, or frequently stopping at breakpoints, and you find debug performance too slow, try turning on **Check heap when stopping** only in those areas you suspect are causing heap errors.

Notes on Check Heap when stopping

- For the Check heap when stopping choice to work, you have to compile your application to use the heap-checking versions of memory management functions. See your compiler reference for more information.
- If you enable the Check heap when stopping choice and run your application to termination, and the application contains a heap error, the heap check is not made. To check the heap just before termination, set a breakpoint on the last line of your application.
- ▶ OS/2 If you are debugging a multiple thread program and a thread stops while running in compiler memory management code that is holding a memory semaphore, the heap check will not be performed.
- ▶ OS/2 If the stopping thread is running in 16-bit code, the heap check will not be performed.

Enabling and disabling a monitored variable, expression or register

You can disable the monitoring of a variable, expression or register. The advantage of disabling a monitored expression instead of deleting it is that it is easier to enable a monitored expression than to recreate it.

You can enable or disable monitored variables, expressions or registers from either the Monitor pane or Locals pane.

To enable or disable a monitored expression:

1. Locate the variable, expression or register you want to disable or enable in the Monitors pane or Locals pane.
2. Right-click on the variable, expression or register you want to enable or disable.
3. Select **Enable** or **Disable** from the pop-up menu.

Enabling hover help for variables

Hover help for variables provides you with 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. This feature is enabled by default when you first start the debugger.

To enable hover help for variables, select **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 that hover help for variables is enabled.

To enable hover help for variables as the default:
1. Select **File > Preferences** from the main menu.
2. Select **Debug** from the list of preferences to set.
3. Select **Allow Tool Tip Evaluation** from the **Debugger Defaults** section.
4. Click **OK** to enable the tool tip monitor and dismiss the dialog.

RELATED CONCEPTS
Distributed Debugger: Monitors

Changing the representation of monitor contents

You can change the representation of variables and expressions in the Monitors pane or Locals pane or the Registers pane. You can change the representation for existing entries or the default representation for future entries in the Applications Preferences dialog.

To change the representation of a variable or expression:
1. Right-click on the variable or expression you want to change the representation of.
2. Select **Representation** from the pop-up menu. The Monitor Representation dialog appears.
3. Select the representation you want from the list of available representations.
4. Click **OK** to change the representation and dismiss the Monitor Representation dialog.

To change the default representation of variables or expressions:
1. Select **File > Preferences** from the main menu bar. The Application Preferences dialog appears.
2. In the left-hand pane of the Application Preferences dialog, go to **Debug >** *program* **>** *language***: Default Monitor Representation**, where *program* is the name of a program loaded in the Distributed Debugger you want to change the default representation for and *language* is the language the program you are debugging was written in.
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 dialog.

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

Appendix A. Distributed debugger dommands

idebug command

The `idebug` command starts both the Distributed Debugger interface and the debug engine when debugging a program locally. When debugging remotely, it is used to connect to a debug engine daemon on a remote system or to start the debugger user interface as a daemon on your local system.

The `idebug` command has the following syntax for AIX or Windows:

```
idebug [idebug_options] [local_debug_parameters | remote_debug_parameters |
ui_daemon_parameters] [–] [program_name [program_parameters]]
```

The `idebug_options` are zero or more of the following:

Option	Purpose
-a `process_id`	Attach to the already running process `process_id`. JAVA Use a `process_id` of `0` (zero) when attaching to a Java Virtual Machine (JVM).
OS/2 -c `child_process_id`	Start debugging the specified child process of the program being debugged. This option only applies to the program you are debugging on OS/2, and is ignored on other platforms.
-h or -?	Display help for the `idebug` command.
-i	Start the debugger in the system initialization code that precedes the call to the main entry point for the program. C++ This can be useful if you need to debug the constructors for static class objects.
-p+	Use program profile information. The debugger will restore window sizes, positions, fonts, and breakpoints for your program from the last time you debugged the program. If you are debugging the program for the first time, the debugger windows start up with their default appearance, and no breakpoints are set. 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.

Option	Purpose
-p-	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. JAVA For Java applications, the debugger also opens the AS/400 Java Console window.
-qquiet	Suppresses the splash screen when the debugger starts.
-s	Prevents the debugger from stopping in the first debuggable statement in the program. Program execution only stops when the first set breakpoint is encountered. This option requires that the program you want to debug has program profile information available. If no program profile information is available, or you specify the `-p-` option, the program will run to completion.

Use the `local_debug_parameters` when you want to start debugging a program on your local system. If a parameter is not specified in the command, the default is assumed.

The `local_debug_parameters` are:

Parameter	Description
JAVA -classpath=`<path>`	This parameter can only be used when debugging interpreted Java programs. Specifies a temporary classpath for the debugger to use to find the Java classes you are debugging. If this is not specified, the debugger uses your present CLASSPATH environment variable setting to find the Java classes you are debugging.
JAVA -qjvmargs= `<jvm_arguments>`	This parameter can only used when debugging interpreted Java programs. Specifies the arguments passed to the JVM that will run the program you want to debug. If you specify more than one option, enclose the list of options in quotation marks. For a list of supported JVM arguments, see the related topic below.
WIN -qlang= `<dominant_language>`	Specifies the dominant language to use for debugging. Valid values for `<dominant_language>` are:

Value:	Use when debugging:
c	C programs
cpp	C++ programs High Performance Compiled (HPC) Java programs
java	Interpreted Java programs

Use the `remote_debug_parameters` when you want to connect to a debug engine daemon on a remote system. If a parameter is not specified in the command, the default is assumed.

The `remote_debug_parameters` are:

Parameter	Description
JAVA -classpath=`<path>`	This parameter can only be used when debugging interpreted Java programs. Specifies a temporary classpath for the debugger to use to find the Java classes you are debugging. If this is not specified, the debugger uses your present CLASSPATH environment variable setting to find the Java classes you are debugging.
-qhost=`<remote_host>`	Specifies the TCP/IP name or address of the machine where the debug engine is running. This parameter is required when debugging remotely.
JAVA -qjvmargs= `<jvm_arguments>`	This parameter can only used when debugging interpreted Java programs. Specifies the arguments passed to the JVM that will run the program you want to debug. If you specify more than one option, enclose the list of options in quotation marks. For a list of supported JVM arguments, see the related topic below.
-qlang= `<dominant_language>`	Specifies the dominant language to use for debugging. These are the valid values for `<dominant_language>`:

Value:	Use when debugging:
c	C programs
cpp	C++ programs High Performance Compiled (HPC) Java programs
java	Interpreted Java programs

Parameter	Description
-qport=`<host_port>`	Specifies the port number on the machine where the debug engine is running. The default port is 8000. This port number must match the port number used in the `-qport` parameter of the `irmtdbgc` command. JAVA This port number must match the port number used in the `-qport` parameter of the `irmtdbgj` command.

The `ui_daemon_parameters` are used when starting 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:

Parameter	Description
-qdaemon	Tells the Distributed Debugger user interface to run as a daemon. You must use the `-quiport` option when specifying `-qdaemon`. If this option is not specified the Distributed Debugger will run locally, not as a daemon.

Parameter	Description	
WIN -qlang= `<dominant_language>`	Specifies the dominant language to use for debugging. Valid values for `<dominant_language>` are:	
	Value:	**Use to when debugging:**
	c	C programs
	cpp	C++ programs High Performance Compiled (HPC) Java programs
	java	Interpreted Java programs
-quiport=`<port>`	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, `<port>` must be a comma-delimited list of port numbers. This option is required when using the `-qdaemon` option. There is no default port number. JAVA One of the port numbers specified here must be used as the port number for the `-qport` parameter of the `irmtdbgc` command. JAVA One of the port numbers specified here must be used as the port number for the `-qport` parameter of the `irmtdbgj` command.	
-qterminate	Closes any running Debugger user interface daemons before starting a new user interface daemon.	

Use the "–" parameter to separate debugger options and parameters from the program name and parameters. Use this option if your program name or parameters include forward slashes ("/") or dashes ("-"). If you do not use this option, anything preceded by a slash or a dash will be interpreted as a debugger option.

If you do not specify `program_name` when issuing the `idebug` command, the debugger will prompt you for the required information in the Load Program dialog.

RELATED TASKS

Starting the debugger for local debugging
Starting the debugger for remote debugging

JAVA Attaching to a running Java Virtual Machine

SOLARIS Starting the debugger on Solaris

RELATED REFERENCES

irmtdbgc command

JAVA irmtdbgj command

JAVA Supported Java Virtual Machine arguments

irmtdbgj command

Restriction: JAVA This command is only used when debugging interpreted Java programs remotely. Use the `idebug` command to debug interpreted Java locally.

Requirement: You must have the debug engine installed on the remote system in order to use this command.

The `irmtdbgj` command starts the Java debug engine on the remote system. Once started, the debug engine waits to connect to the debugger user interface on the local system. You can use the debugger Load Program on the local system or use the `idebug` command with the remote debug parameters to start the debugger interface on the local system.

The `irmtdbgj` command has the following syntax:

```
irmtdbgj [irmtdbgj_options] [JVM_attach_parameters]
[engine_daemon_parameters |ui_daemon_parameters [class_name
[class_parameters]]]
```

where `irmtdbgj_options` are zero or more of the following:

Parameter	Description
-help	Displays help for the `irmtdbgj` command.
-qquiet	Suppresses console output.

Use the `JVM_attach_parameters` if you want to attach the debugger to a running Java Virtual Machine (JVM). You can attach to a JVM if you are starting a debug engine daemon or connecting to a debugger user interface daemon. The `jvm_attach_parameters` are:

Parameter	Description
-password= `<agent_password>`	Specifies the agent password. This password is printed when you start your java program using the one of the following commands: • for Java 1.1.x, use the `java_g -debug` command. • on the Java 2 platform, use the `java -Xdebug` command. This parameter is required when attaching to a running JVM.
-host= `<JVM_host>`	Specifies the name or address of the machine where the JVM is running. If the JVM is running on the same machine where the debug engine will run, use “localhost”.

Use zero or more of the `engine_daemon_ parameters` if you want to start the debug engine as a daemon on the remote system. The debug engine daemon listens on a port number for a connection from the debugger user interface. Once a connection is made, you can begin debugging your program from the local system. The `engine_daemon_parameters` are:

Parameter	Description
-multi	This option enables multiple session debugging. If this option is omitted, only single session debugging is enabled. In single session debugging, the debug engine terminates when the program you are debugging runs to completion or is terminated manually. In multiple session debugging, the debug engine re-initializes itself and waits for a new connection when the program you are deubugging runs to completion or is terminated manually. The debug engine must be terminated manually on the remote system.
-qjvmargs= `<jvm_arguments>`	Specifies arguments passed to the JVM that will run the program you want to debug. If you specify more than one option, eclose the list of options in quotation marks. **Attention:** Using unsupported arguments will cause the debugger to exit. For a list of supported JVM arguments, see the related topic below.
-qport=`<port>`	Specifies the TCP/IP port where the debug engine daemon will listen for the user interface. The default port is 8000. If you do not use the default port, specify the same port number you use here in the `-qport` parameter of the `idebug` command.

Use the `ui_daemon_parameters` if you have started the debugger user interface as a daemon. As daemon, the debugger user interface listens on a port for a debugger engine connection. Upon connecting, the debugger user interface starts and you can begin debugging your program. The `ui_daemon_parameters` are:

Parameter	Description
-qhost=`<ui_daemon_host>`	Specifies the IP address or name of the machine where the user interface daemon is running. This parameter is required when connecting to a debugger user interface daemon.
-qtitle=`<ui_daemon_title>`	Specifies the title that will appear on the process tab
-quiport=`<ui_daemon_port>`	Specifies the TCP/IP port used for the connection. The default port is 8001. If you do not use the default port, specify the same port number you use here in the `-quiport` parameter of the `idebug` command.

If you do not specify `class_name` when issuing the `irmtdbgj` command with the `ui_daemon_parameters`, the debugger will prompt you for the required information in the Load Program dialog of the debugger user interface.

RELATED TASKS
Starting the debugger for remote debugging
Starting the debugger interface daemon
JAVA Attaching to a running Java Virtual Machine

RELATED REFERENCES
idebug command
JAVA Supported Java Virtual Machine arguments

Supported Java Virtual Machine arguments

Advanced Java Virtual Machine (JVM) arguments are available when debugging your interpreted Java programs. The arguments are passed to the debugger in the following ways:

- in the Load Program dialog for Java
- with the `-qjvmargs` option of the `idebug` command
- with the `-qjvmargs` option of the `irmtdbgj` command

The following JVM arguments are supported when debugging your interpreted Java 1.1.x programs:

JVM argument	Description
-noasyncgc	Disables asynchronous garbage collection.
-noclassgc	Disables class garbage collection.
-ss`<memory_size>`	Sets the maximum amount of memory allocated to the native stack for any thread.
-oss`<memory_size>`	Sets the maximum amount of memory allocated to Java stack for any thread.
-ms`<memory_size>`	Sets the initial amount of memory allocated to the Java heap.
-mx`<memory_size>`	Sets the maximum amount of memory allocated to the Java heap.
-classpath `<directory_list>`	Sets the list of directories in which to search for classes. This list replaces any classpaths you may have already set up for this JVM only. OS/2 WIN Directories must be separated by semi-colons. AIX SOLARIS HP-UX Directories must be separated by colons. **Note:** There is one space between the `-classpath` argument and the list of directories to search.
-prof[:`<file>`]	Outputs profiling data to a file called `java.prof` in the current directory, or to a file name specifed.
-verify	Verify all classes when read in.
-verifyremote	Verify classes read in over the network. This is a default argument.
-noverify	Do not verify any classes.
-nojit	Disables the JIT compiler.

The following JVM arguments are supported when debugging your program on the Java 2 platform:

JVM argument	Description
-Xnoclassgc	Disables class garbage collection.

JVM argument	Description
-Xms`<memory_size>`	Sets the initial amount of memory allocated to the Java heap.
-Xmx`<memory_size>`	Sets the maximum amount of memory allocated to the Java heap.
-Xbootclasspath: `<directory_list>`	Sets the list of directories in which to search for classes. This list replaces any classpaths you may have already set up for this JVM only. WIN Directories must be separated by semi-colons. AIX SOLARIS HP-UX Directories must be separated by colons. **Note:** There is no space between the `-Xbootclasspath:` argument and the list of directories to search.
-Djava.compiler=NONE	Disables the JIT compiler.

RELATED REFERENCES
idebug command
irmtdbgj command

irmtdbgc command

Restriction: This is supported on AIX and Windows only.

Requirement: You must have the debug engine installed on the remote system in order to use this command. Check the install documentation for instructions on how to install the debug engine on a remote system.

The `irmtdbgc` command starts the debug engine on the remote system. If the debug engine detects a debugger user interface daemon, then you can start debugging your program immediately. If no debugger user interface daemon is detected, the debug engine will run as a daemon until you start the debugger user interface on the local system with the `idebug` command.

The `irmtdbgc` command has the following syntax:

```
irmtdbgc [irmtdbgc_parameters] [–] [program_name [program_parameters]]
```

where `irmtdbgc_parameters` are:

Parameter	Description
-qprotocol=`<protocol>`	Specifies the communications protocol to use. Only TCP/IP is supported. This is the default protocol.

Parameter	Description
-qport=`<port>`	Specifies the TCP/IP port used for the connection. If you do not use the default port, specify the same port number you use here in the `-qport` parameter of the `idebug` command. The default port is 8000. **Restriction:** Do not use this parameter when connecting to a debugger user interface daemon. You must use the `-quiport` option.
-quiport=`<ui_daemon_port>`	Specifies the TCP/IP port used for connecting to a debugger user interface daemon listening on another machine. This port number must match the port number used in the `-quiport` parameter of the `idebug` command.
-qhost=`<ui_daemon_host>`	Specifies the TCP/IP name or address of the machine where the debugger user interface daemon is listening.
-qsession=`single|multi`	Specifies whether to support single session debugging or multiple session debugging. The default is `single`. In single session debugging, the debug engine terminates when the program you are debugging runs to completion or is terminated manually. In multiple session debugging, the debug engine re-initializes itself and waits for a new connection when the program you are deubugging runs to completion or is terminated manually. The debug engine must be terminated manually on the remote system.

Use the "–" parameter to separate `irmtdbgc` parameters from the program name and parameters. Use this option if your program name or parameters include forward slashes ("/") or dashes ("-"). If you do not use this option, anything preceded by a slash or a dash will be interpreted as a debugger option.

If you do not specify `program_name` when issuing the `irmtdbgc` command, the debugger will prompt you for the required information in the Load Program dialog of the debugger user interface.

Tip: If this command gives you an error, you may have an older version of the debug engine on your remote system. Install a new version of the debug engine, if available. If no newer version of the debug engine is available, try using the `irmtdbg` command.

RELATED REFERENCES
idebug command

idod command

WIN **Restriction:** Debug on demand is available only on Windows. It is only available for local debugging.

The `idod` command enables and disables the debug on demand feature of the debugger. The `idod` command has the following syntax:

```
idod [idod_options]
```

where `idod_options` are one of the following:

Option	Purpose
`idebug`	Enable the debug on demand feature of the debugger.
`/u`	Disable the debug on demand feature of the debugger.

RELATED CONCEPTS

WIN Debug on demand

RELATED TASKS

WIN Enabling and disabling debug on demand

Appendix B. Compiler options for debugging

Interpreted Java compiler options

If you use the **javac** compiler to compile your code for debugging, you can set breakpoints and step through your source code without using any compiler options. Use the **-g** option if you want to examine local, class instance and static variables while debugging.

Here is a partial list of compiler options to consider when compiling your classes:

Option	**Purpose**
-g	Compiles your code with debug information. Use this option if you want to examine the contents of local variables when debugging your classes. You can still set breakpoints and step through your code if you do not compile your classes with this option.
-O	Compiles and optimizes your code. Do not use this option if you want to debug your classes. If you compile your code with this option **all** debugging information is removed from the class during optimization.
-classpath `<path>`	Overrides the **CLASSPATH** environment variable with the path specified by `<path>`. Use this option when you want to try compiling something without modifying the **CLASSPATH** environment variable.
-d `<dir>`	Determines the root directory where compiled classes are stored. This is useful since classes are often organized in a hierarchical directory structure. With this option, the directories are created below the directory specified by `<dir>`.

For a complete list of compiler options, refer to documentation provided with the JDK.

Appendix C. Supported expressions

Interpreted Java expressions supported

Only expressions using the dot (.) operator are supported. The dot operator is used to access instance and static variables within objects and classes.

C/C++ expressions supported

C/C++ supported data types

You can monitor an expression that includes a cast to any of the following types:

- 8-bit signed char
- 8-bit unsigned char
- 16-bit signed integer
- 16-bit unsigned integer
- 32-bit signed integer
- 32-bit unsigned integer
- 64-bit signed integer
- 64-bit unsigned integer
- 32-bit floating-point
- 64-bit floating-point
- Pointers
- User-defined types

These data types include **int**, **short**, **char** and so on.

C/C++ supported expression operands

You can monitor an expression that uses the following types of operands only:

Operand	**Definition**
Variable	A variable used in your program.
Constant	The constant can be one of the following types: • Fixed-point or floating-point constant within the ranges supported by the system the program you are debugging is running on. • A string constant, enclosed in double quotation marks (for example, `"mystring"`) • A character constant, enclosed in single quote marks (for example, `’x’`)

Operand	Definition
AIX OS/2 WIN Register	Any of the processor registers that can be displayed in the Registers Monitor. In the case of conflicting names, program variable names take precedence over register names. For conversions that are done automatically when the registers display in mixed-mode expressions, general-purpose registers are treated as unsigned arithmetic items with a length appropriate to the register. For example, on Intel platforms EAX is 32-bits, AX is 16-bits, and AL is 8-bits.

If you monitor an enumerated variable, a comment appears to the right of the value. If the value of the variable matches one of the enumerated types, the comment contains the name of the first enumerated type that matches the value of the variable. If the length of the enumerated name does not fit in the monitor, the contents appear as an empty entry field.

The comment (empty or not) lets you distinguish between a valid enumerated value and an invalid value. An invalid value does not have a comment to its right.

You *cannot* update an enumerated variable by entering an enumerated type. You must enter a value or expression. If the value is a valid enumerated value, the comment to the right of it is updated.

You cannot look at variables that have been defined using the `#define` preprocessor directive.

C/C++ supported expression operators

You can monitor an expression that uses the following operators only:

Operator	Coded as
C++ Global scope resolution	::*a*
C++ Class or namespace scope resolution	*a*::*b*
Subscripting	*a*[b]
Member selection	*a*.*b* or *a*->*b*
Size	`sizeof` *a* or `sizeof` (*type*)
Logical not	!*a*
Ones complement	~*a*
Unary minus	-*a*
Unary plus	+*a*
Dereference	**a*
Type cast	(*type*) *a*
Multiply	*a* * *b*
Divide	*a* / *b*
Modulo	*a* % *b*
Add	*a* + *b*

Operator	**Coded as**
Subtract	*a* - *b*
Left shift	*a* << *b*
Right shift	*a* >> *b*
Less than	*a* < *b*
Greater than	*a* > *b*
Less than or equal to	*a* <= *b*
Greater than or equal to	*a* >= *b*
Equal	*a* == *b*
Not equal	*a* != *b*
Bitwise AND	*a* & *b*
Bitwise OR	*a* | *b*
Bitwise exclusive OR	*a* ˆ *b*
Logical AND	*a* && *b*
Logical OR	*a* || *b*

Appendix D. Environment variables

PATH environment variable

The PATH environment variable is used to locate the debugger executable and the executable programs to be debugged, as well as any other executables being run on the workstation.

WIN the PATH environment variable is also used to locate DLLs.

OS/2 For OS/2, PATH and LIBPATH provide equivalent functionality.

RELATED TASKS
Setting environment variables for the debugger

RELATED REFERENCES

AIX OS/2 LIBPATH environment variable

DPATH environment variable

The DPATH environment variable is used to locate message files, which the debugger needs to display messages and the text of menus and dialogs.

RELATED TASKS
Setting environment variables for the debugger

RELATED REFERENCES

AIX OS/2 LIBPATH environment variable

CLASSPATH environment variable

JAVA The CLASSPATH environment variable tells the debugger, as well as the Java Virtual Machine and other Java applications, where to find your class libraries.

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

RELATED TASKS
Setting environment variables for the debugger

INCLUDE envrionment variable

The INCLUDE environment variable is used by the debugger to locate include files on the workstation.

The environment variable does not apply to languages that do not support include files.

RELATED TASKS
Setting environment variables for the debugger

LIBPATH environment variable

AIX OS/2 **Restriction:** This is supported on AIX and OS/2 only.

The LIBPATH environment variable tells the debug engine where to look for debugger DLLs on the workstation.

OS/2 The LIBPATH environment variable must be set from within CONFIG.SYS; you cannot set it using the SET command.

OS/2 For OS/2, LIBPATH and PATH provide equivalent functionality.

RELATED REFERENCES
Other environment variables

DER_DBG_CASESENSITIVE envrionment variable

The DER_DBG_CASESENSITIVE environment variable, if set to a non-null value (for example, "yes", 1, "true", etc.) tells the debugger to compare part names and module names on a case-sensitive basis. By default the debugger converts all names to uppercase for comparison purposes. Note that this does not affect filesystem accesses which are operating system dependent and not affected by DER_DBG_CASESENSITIVE.

RELATED TASKS
Setting environment variables for the debugger

DER_DBG_LOCAL_PATH environment variable

Restriction: This is supported on AIX and Windows only.

AIX WIN The DER_DBG_LOCAL_PATH environment variable is used to locate executables and DLLs on the system where you are debugging your program.

OS/2 For OS/2, LIBPATH and PATH provide equivalent functionality.

RELATED TASKS
Setting environment variables for the debugger

DER_DBG_NUMBEROFELEMENTS environment variable

The DER_DBG_NUMBEROFELEMENTS environment variable can be set to an integer value to tell the debugger the maximum number of elements to display for an array, structure, or object in a Program or Storage monitor.

RELATED TASKS
Setting environment variables for the debugger

DER_DBG_OVERRIDE environment variable

Restriction: This is supported on AIX and Windows only.

The DER_DBG_OVERRIDE environment variable takes precedence over DER_DBG_PATH. If you set your DER_DBG_PATH variable in your system settings, but you want to temporarily add another path that takes precedence over DER_DBG_PATH, set DER_DBG_OVERRIDE. To restore DER_DBG_PATH as the path used to locate executables and DLLs, clear DER_DBG_OVERRIDE. You can clear DER_DBG_OVERRIDE using the following command:

```
set DER_DBG_OVERRIDE=
```

RELATED TASKS
Setting environment variables for the debugger

RELATED REFERENCES
DER_DBG_PATH environment variable

DER_DBG_PATH environment variable

The DER_DBG_PATH environment variable is used by the debug engine, the debugger user interface, or both to locate debug source files on your client workstation that are not stored in the same location as the executable being debugged. For example, if your debug executable is stored in F:\BUILDS\SANDDUNE\TEST but your source code is stored in F:\SOURCE and F:\SOURCE\INCLUDE , you should set your DER_DBG_PATH variable as follows:

```
set DER_DBG_PATH=F:\SOURCE;F:\SOURCE\INCLUDE
```

You can set the DER_DBG_PATH environment variable on both client and server systems. The search for source files starts on the server first.

Note: The DER_DBG_PATH environment variable must be set on both the client and server systems before starting the Distributed Debugger user interface daemon.

RELATED TASKS
Setting environment variables for the debugger

DER_DBG_TAB environment variable

The DER_DBG_TAB environment variable affects how the debugger expands tab characters in a source or mixed view within a Source pane. The value for this variable is an integer, indicating the number of spaces to convert a tab character into. Unlike DER_DBG_TABGRID, DER_DBG_TAB does not cause the debugger to place tabbed information in specific columns; it simply results in each tab in the displayed files being converted to the indicated number of spaces.

Note: If DER_DBG_TABGRID has been set to a nonzero value, the setting of DER_DBG_TAB has no effect.

RELATED TASKS
Setting environment variables for the debugger

RELATED REFERENCES
DER_DBG_TABGRID environment variable

DER_DBG_TABGRID environment variable

The DER_DBG_TABGRID environment variable affects how the debugger uses tab characters to align tabs to columns in a source or mixed view within a Source pane. The value of this variable is an integer indicating the starting position and frequency of the tab. For example, if you set DER_DBG_TABGRID=6, the debugger sets tab stops at 6, 12, 18, 24, and so on. If DER_DBG_TABGRID is set to a nonzero value, the setting of DER_DBG_TAB has no effect.

RELATED TASKS
Setting environment variables for the debugger

RELATED REFERENCES
DER_DBG_TAB environment variable

Appendix E. Limits

Remote debug limitations

Remote debugging imposes the following limitations:

- **Browse** only displays the file system on the local system. The file system on the remote system cannot be displayed.

RELATED CONCEPTS
Remote debugging

RELATED TASKS
Starting the debugger for remote debugging

Interpreted Java debug limitations

Limitations exist in the JDK 1.1.X or lower that may cause problems when debugging your interpreted Java classes. Be aware of the following limitations and problems when debugging:

- Console input is not accepted.
- You cannot suspend, start, or stop threads.
- Stepping behavior may be erratic when stepping into constructors, or when stepping into or over `SystemLoad` library functions.
- The debugger cannot halt an applet or application that has all of its threads blocked.
- You cannot modify the contents of monitored variables.
- Breakpoints set on static initializers or static blocks will be ignored.
- Breakpoints set on `try` statements are ignored.
- Execution will not stop inside catch blocks for thrown errors.
- The debugger will not notify the user when classes extending from `java.lang.Error` are thrown. The debugger will notify the user when classes extending from `java.lang.Exception` are thrown.
- When debugging JAR files, the source code for classes contained in a JAR file must be available outside of the JAR file.
- After exiting a program block, variables now out of scope may still be shown in a monitor.
- Debug-agent error messages may appear intermittently while debugging your classes.
- If you step into a class that has not been registered with the debugger, you may receive a **"Cannot find source for null"** message**.** If this happens, issue a step return command to continue debugging. To avoid this problem, register the appropriate class source file or package containing the class source with the debugger before you start debugging.
- When debugging remotely, communication between the debugger and the program being debugged may be terminated prematurely by the JVM.

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

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.