Debugging Java Applications

Table of Contents

NetBeans IDE provides a rich environment for troubleshooting and optimizing your applications. Built-in debugging support allows you to step through your code incrementally and monitor aspects of the running application, such as values of variables, the current sequence of method calls, the status of different threads, and the creation of objects.

When using the IDE's debugger, there is no reason for you to litter your code with `System.out.println` statements to diagnose any problems that occur in your application. Instead, you can use the debugger to designate points of interest in your code with breakpoints (which are stored in the IDE, not in your code), pause your program at those breakpoints, and use the various debugging windows to evaluate the state of the running program.

In addition, you can change code while debugging and dynamically reload the class in the debugger without having to restart the debugging session.

The following are some of the things that you can do within the IDE's debugger:

- Step through application code line by line.
- Step through JDK source code.
- Execute specific chunks of code at a time (using breakpoints as delimiters).
- Suspend execution when a condition that you have specified is met (such as when an iterator reaches a certain value).
- Suspend execution at an exception, either at the line of code that causes the exception or in the exception itself.
- Track the value of a variable or expression.
- Track the object referenced by a variable (fixed watch).
- Fix code on the fly and continue the debugging session.
- Suspend threads individually or collectively.
- Step back to the beginning of a previously called method (pop a call) in the current call stack.
- Run multiple debugging sessions at the same time. For example, you might need this capability to debug a client/server application.

Starting a Debugging Session

The simplest way to start using the debugger is to choose Run | Step Into. The program counter (marked by green background highlighting and the ⇨ icon as shown in Figure 5-1) stops one line into the main method of your main project.

You can then step through your code incrementally with any of the Step commands to observe the program flow and monitor the evolving values of variables in the Local Variables window. See Stepping Through Code on page XXX for a description of all of the Step commands and the ensuing topics for information on how to take advantage of the debugger's capabilities.

NetBeans IDE Tip

You can also use the Run To Cursor command to start a debugging session. In the Source Editor, click in the line where you want execution to initially suspend and choose Run | Run To Cursor. This command only works for starting a debugging session if you select a line of code in the project's main class or a class directly called by the main class in the main project.

Figure 5-1
A suspended program with the green program counter showing the next line to be executed

More likely, you will want to start stepping through code at some point after the start of the main method. In this case, you can specify some point in the program where to initially suspend the debugged execution and then start the debugger. To do this:

1. Set a line breakpoint in your main project by opening a class in the Source Editor and clicking in the left margin next to the line where you want to set the breakpoint (or pressing Ctrl-F8)..

    You know that the breakpoint has been set when the pink ▫ glyph appears in the margin and the line has pink background highlighting (as shown in Figure 5-2).

Figure 5-2

Code in the Source Editor with a debugger breakpoint set

2. Press F5 to start debugging the main project.

When the execution of the program stops at the breakpoint (which you can see when pink breakpoint highlight is replaced by the green highlight of the program counter), you can step through the code line by line while viewing the status of variables, threads, and other information.

See the ensuing topics for details on stepping and viewing program information.

NetBeans IDE Tip
If you have set up a freeform project, you need to do some extra configuration to get the debugging commands to work. See Chapter 12 for more details.

Debugger Windows

When you start debugging a program, the Debugger Console appears as a tab in the lower left corner of the IDE (as shown in Figure 5-3). The Debugger Console logs the execution status of the debugged program (such as whether the code is stopped at a breakpoint). In addition, a tab opens in the Output to log any application output (as well the output from the Ant build script the IDE uses when running the command).

In the lower right hand corner, several windows (Watches, Local Variables, and Call Stack) open as tabs and provide current information on the debugging session, such as the current values of variables and a list of current method calls. You can also open individual debugging windows by choosing from the Windows | Debugging menu.

Figure 5-3
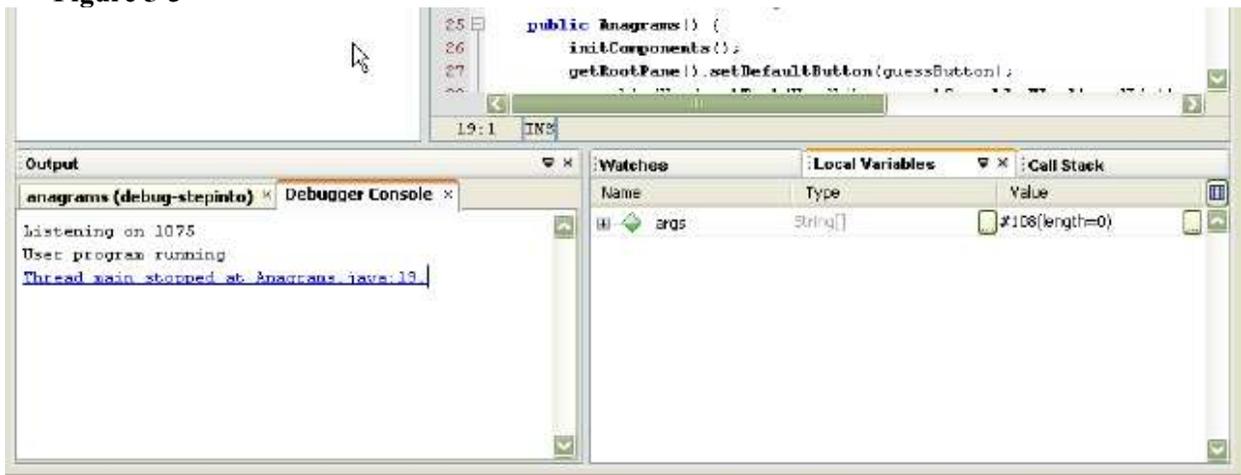
Windows that appear when you start debugging in the IDE, including the Debugger Console, and windows for Watches, Local Variables, and the Call Stack

Most of the windows display values according to the debugger's current *context*. In general, the current context corresponds to one method call in one thread in one session. You can change the context (for example, designate a different current thread in the Threads window) without affecting the way the debugged program runs.

See Table 5-1 for a list of all of the windows available and how to open them.

Table 5-1: Debugger windows

Debugger Window	Open With	Description
Local Variables	Alt-Shift-1 (or Window | Debugging | Local Variables)	Displays all fields and local variables in the debugger's current context and their current values. Fields are listed under the `this` node.
Watches	Alt-Shift-2 (or Window | Debugging | Watches)	Displays the names of fields, local variables, or expressions that you have placed a watch on. Though all of your watches are displayed no matter the current context, the value displayed is the value for that context (not for the context that the watch was set in). For example, if you have a watch on the `this` keyword, the `this` referred to in the Watches window will always correspond to the object referred to from the current method call.
Call Stack	Alt-Shift-3 (or Window | Debugging | Call Stack)	Displays all method calls in the current chain of calls. The Call Stack window enables you to jump directly to code of a method call, back up the program's execution to a previous method call, or select a context for viewing local variable values.

Debugger Window	Open With	Description
Classes	Alt-Shift-4 (or Window | Debugging | Classes)	Provides a tree view of classes for the currently debugged application grouped by classloader.
Breakpoints	Alt-Shift-5 (or Window | Debugging | Breakpoints)	Displays all breakpoints that you have set in all running debugging sessions.
Threads	Alt-Shift-6 (or Window | Debugging | Threads)	Displays the threads in the current session. In this window, you can switch the context by designated another thread is the current thread.
Sessions	Alt-Shift-7 (or Window | Debugging | Sessions)	Displays a node for each debugging session in the IDE. From this window, you can switch the current session.
Sources	Alt-Shift-8 (or Window | Debugging | Sources)	Displays sources that are available for debugging and enables you to specify which ones to use. For example, you can use this window to enable debugging with JDK sources.

Attaching the Debugger to a Running Application

If you need to debug an application that is running on another machine or is running in a different virtual machine, you can attach the IDE's debugger to that application:

1. Start the application that you are going to debug in debug mode. This entails adding some special arguments to the script that launches the application.

   For the Windows users using a Sun JDK, the argument list might look like the following (all in one line and no space after `-Xrunjdwp:`):

   ```
   java -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:
   transport=dt_shmem,server=y,address=MyAppName,suspend=n
   -classpath C:\my_apps\classes mypackage.MyApp
   ```

   On other operating systems (or on a Windows machine when you are debugging an application running on a different machine), the argument list might look something like the following:

   ```
   java -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:
   transport=dt_socket,server=y,address=8888,suspend=n
   -classpath HOME/my_apps/classes mypackage.MyApp
   ```

   See Table 5-2 for a key to these options. For more complete documentation of the options, visit http://java.sun.com/products/jpda/doc/conninv.html.

2. In the IDE, open the project that contains the source code for the application to be debugged.

3. Choose Run | Attach Debugger.

4. In the Attach dialog box, select the connector from the Connector combo box.

   Choose SharedMemoryAttach if you want to attach to an application that has been started with the `dt_shem` transport. Choose SocketAttach if you want to attach to an application that has been started with the `dt_socket` transport.

   See the Connectors row of Table 5-3 for information on the different types of connectors.

5. Fill in the rest of the fields. The fields that appear after Connector depend upon the kind of connector that you have selected. See Table 5-3 for a key to the different fields.

Table 5-2: Debugger Launch Parameters

Launch Parameter or Sub-parameter	Description
-Xdebug	Enables the application to be debugged.
-Xnoagent	Disables the `sun.tools.debug` so that the JPDA debugger can properly attach its own agent.
-Djava.compiler	Disables the JIT (Just-In-Time) compiler.
-Xrunjdwp	Loads the reference implementation of the Java Debug Wire Protocol, which enables remote debugging.
transport	Name of the transport to be used when debugging the application. The value can be `dt_shmem` (for a shared memory connection) or `dt_socket` (for a socket connection). Shared memory connections are only available on Windows machines.
server	If this value equals `n`, the application attempts to attach to the debugger at the address specified in the `address` sub-parameter. If this value equals `y`, the application listens for a connection at this address.
address	For socket connections, specifies a port number used for communication between the debugger and the application. For shared memory connections, specifies a name that refers to the shared memory to be used. This name can consist of any combination of characters that are valid in file names on a Windows machine except the backslash. You then use this name in the Name field of the Attach dialog box when you attach the debugger to the running application.
suspend	If the value is `n`, the application starts immediately. If the value is `y`, the application waits until a debugger has attached to it before executing.

Table 5-3: Attach Dialog Box Fields

Field	Description
Connector	Specifies the type of JPDA connector to use. On Windows machines, you can choose between shared memory connectors and socket connectors. On other systems, you can only use a socket connector. For both shared memory connectors and socket connectors, there are Attach and Listen variants. You can use an Attach connector to attach to a running application. You can use a Listen connector if you want the running application to initiate the connection to the debugger. If you use the Listen connector multiple applications running on different JVMs can connect to the debugger.
Transport	Specifies the JPDA transport protocol to use. This field is automatically filled in according to what you have selected in the Connector field.
Host	(Only for socket attach connections.) The host name of the computer where the debugged application is running.
Port	(Only for socket connections.) The port number that application attaches to or listens on. You can assign a port number in the `address` sub-parameter of the `Xrunjdwp` parameter that you pass to the JVM of the application that is to be debugged. If you do not use this sub-option, a port number is assigned automatically, and you can determine the assigned port number by looking at the output of the process.
Timeout	The number of seconds that the debugger waits for a connection to be established.
Name	(Only for shared memory connections.) Specifies the shared memory to be used for the debugging session. This value must correspond to the value of the `address` sub-parameter of the `Xrunjdwp` parameter that you pass to the JVM of the application that is to be debugged.
Local Address	(Only for socket listen connections.) The host name of the computer that you are running on.

Starting the Debugger Outside of the Project's Main Class

If you have multiple executable classes in your project, there might be times when you want to start the debugger from a class different than the one that is specified as the project's main class.

To start the debugger on a class other than the project's main class, right-click the file's node in the Projects window or Files window and choose Debug File.

You can only start the debugger on a file if it has a `main` method.

Stepping Through Code

Once execution of your program is paused, you have several ways of resuming execution of the code. You can step through code line by line (Step In) or in greater increments. See Table 5-4 for the commands available for stepping or continuing execution or the following sub-topics for a task-based look at the commands.

Table 5-4: Debugger Step Commands

Step Command	Description
Step Into (F7)	Executes the current line. If the line is a call to a method or constructor and there is source available for the called code, the program counter moves to the declaration of the method or constructor. Otherwise, the program counter moves to the next line in the file.
Step Over (F8)	Executes the current line and moves the program counter to the next line in the file. If the executed line is a call to a method or constructor, the code in the method or constructor is also executed.
Step Out of (Alt-Shift-F7)	Executes the rest of the code in the current method or constructor and moves the program counter to the line after the caller of the method or constructor. This command is useful if you have stepped into a method that you do not need to analyze.
Run to Cursor (F4)	Executes all of the lines in the program between the current line and the insertion point in the Source Editor.
Pause	Stop all threads in the current session.
Continue (Ctrl-F5)	Resume execution of the program until the next breakpoint.

Executing Code Line By Line

You can have the debugger step a line at a time by choosing Run | Step Into (F7). If you use the Step Into command on a method call, the debugger enters the method and pauses at the first line, unless the method is part of a library that you have not specified for use in the debugger. See Stepping Into the JDK and Other Libraries on page XXX for details on making sources available for use in the debugger.

Executing a Method Without Stepping Into It

You can execute a method without having the debugger pause within the method by choosing run | Step Over (F8). After you use, the Step Over command, the debugger pauses again at the line after the method call.

Resuming Execution Through the End of a Method

If you have stepped into a method that you do not need to continue analyzing, you can have the debugger complete execution of the method and then pause again at the line after the method call.

To complete execution of a method in this way, choose Run | Step Out Of (Alt-Shift-F7).

Continuing to the Next Breakpoint

If you do not need to observe every line of code while you are debugging, you can continue execution until the next point or until execution is otherwise suspended.

To continue execution of a program that has been suspended at a breakpoint, choose Run | Continue or press Ctrl-F5.

Continuing to the Cursor Position

When execution is suspended, you can continue to a specific line without setting a breakpoint by placing the cursor in that line and choose Run | Run to Cursor (F4).

Stepping Into the JDK and Other Libraries

When you are debugging, you can step into the code for the JDK and any other libraries if you have the source code that is associated with them registered in the IDE's Library Manager. See Chapter 3, Making External Sources and Javadoc Available in the IDE for information on associating source code with a library.

By default, the IDE does not step into JDK sources when you are debugging. If you use the Step In command on a JDK method call, the IDE executes the method and returns the program counter to the line after the method call (as if you used the Step over command).

To enable stepping into JDK sources for a debugged application:

1. Start the debugger for the application.
2. Open the Sources window by choosing Window | Debugging | Sources or by pressing Alt-Shift-8.
3. Select the Use For Debugging checkbox for the JDK.

Figure 5-4
Sources window

Limiting the Classes That You Can Step Into For a Library

If you are using a library for debugging, you can set a filter to exclude some of the sources from being used.

To exclude classes from being used in the debugger:

1. Start the debugger for the application.
2. Open the Sources window by choosing Window | Debugging | Sources or by pressing Alt-Shift-8.
3. Right-click the line for the library that you want to create an exclusion filter for and choose Add Class Exclusion Filter.
4. Type a filter in the Add Class Exclusion Filter dialog box.

The filter can be:

- a fully-qualified class name.
- a package name or class name with an asterisk (*) at the end to create a wildcard. For example, you could type the following to exclude all classes in the `javax.swing` package: `javax.swing.*`
- an expression with a wildcard at the beginning. For example, to exclude all classes that have Test at the end of their names, you could use: `*Test`

You can create multiple class exclusion filters.

To disable the filter, deselect the Use in Debugging checkbox next to the filter in the Sources window.

To delete a class exclusion filter, right-click the filter and choose Delete.

Setting Breakpoints

A breakpoint is a marker that you can set to specify where execution should pause when you are running your application in the IDE's debugger. Breakpoints are stored in the IDE (not in your application's code) and persist between debugging sessions and IDE sessions.

When execution pauses on a breakpoint, the line where execution has paused is highlighted in green in the Source Editor and a message is printed in the Debugger Console with information on the breakpoint that has been reached.

In their simplest form, breakpoints provide a way for you to pause the running program at a specific point so that you can:

- Monitor the values of variables at that point in the program's execution.
- Take control of program execution by stepping through code line by line or method by method.

However, you can also use breakpoints as a diagnostic tool to do things such as:

- Detect when the value of a field or local variable is changed (which, for example, could help you determine what part of code assigned an inappropriate value to a field).
- Detect when an object is created (which might, for example, be useful when trying to track down a memory leak).

You can set multiple breakpoints and you can set different types of breakpoints. The simplest kind of breakpoint is a line breakpoint, where execution of the program stops at a specific line. You can also set breakpoints on other situations, such as the calling of a method, the throwing of an exception, or the changing of a variable's value. In addition, you can set conditions in some types of breakpoints so that they suspend execution of the program only under specific circumstances. See Table 5-5 for a summary of the types of breakpoints.

Table 5-5: Breakpoint Categories

Breakpoint Type	Description
Line	Set on a line of code. When the debugger reaches that line, it stops before executing the line. The breakpoint is marked by pink background highlighting and the ▢ icon. You can also specify conditions for line breakpoints.
Class	Execution is suspended when the class is referenced from another class and before any lines of the class with the breakpoint are executed.
Exception	Execution is suspended when an exception occurs. You can specify whether execution stops on caught exceptions, uncaught exceptions, or both.
Method	Execution is suspended when the method is called.
Variable	Execution is suspended when the variable is accessed. You can also configure the breakpoint to have execution suspended only when the variable is modified.
Thread	Execution is suspended whenever a thread is started or terminated. You can also set the breakpoint on the thread's death (or both the start and death of the thread).

Setting a Line Breakpoint

To set a line breakpoint, click the left margin of the line where you want to set the breakpoint or click in the line and press Ctrl-F8.

To delete the breakpoint, click the left margin of the line or click in the line and press Ctrl-F8.

If you want to customize a line breakpoint, you can do so through the Breakpoints window. Choose Window | Debugging | Breakpoints or press Alt-Shift-5. In the Breakpoints window, right-click the breakpoint and choose Customize.

Setting a Breakpoint on a Class Call

You can set a breakpoint on a class so that the debugger pauses when code from the class is about to be accessed and/or when the class is unloaded from memory.

To set a breakpoint on a class:

1. Choose Run | New Breakpoint (Ctrl-Shift-F8).
2. In the New Breakpoint dialog box, select Class from the Breakpoint Type combo box.
3. Enter the class and package names. These fields should be filled in automatically with the class currently displayed in the Source Editor.

NetBeans IDE Tip

You can specify multiple classes for the breakpoint to apply to, either by using wildcards in the Package Name and Class Name fields, or by selecting the Exclusion Filter checkbox.

Use the asterisk to create wildcards in the Package Name and Class Name fields if you want the breakpoint to apply to multiple classes or all classes in a package. For example, if you enter just an asterisk (*) in the Class Name field, the breakpoint will apply to all classes in the package specified in the Package Name field.

Use the Exclusion Filter checkbox if you want the breakpoint to apply at all classes (including JDK classes) except for the ones that match the

classes or packages specified in the Package Name and Class Name fields. You can set multiple breakpoints with the Exclusion Filter on. For example, you might set an exclusion filter on `com.mydomain.mypackage.mylib.*` because you want the class breakpoint to apply to all of your classes except those in the package `mylib`. However, if you do not want the debugger to pause at the loading of each JDK class that is called, you should also set a class breakpoint with an exclusion filter on `java.*`.

Setting a Breakpoint on a Method or Constructor Call

You can set a breakpoint so that the debugger pauses when a method or constructor is called before any lines of the method or constructor are executed.

To set a breakpoint on a method on constructor:

1. Choose Run | New Breakpoint (Ctrl-Shift-8).
2. In the New Breakpoint dialog box, select Method from the Breakpoint Type combo box.
3. Enter the class, package, and method names. These fields are filled in automatically according to the class open in the Source Editor and the location of the insertion point.

You can make the breakpoint apply to all methods and constructors in the class by checking the All Methods For Given Classes dialog box.

Setting a Breakpoint on an Exception

You can set a breakpoint so that the debugger pauses when an exception is thrown in your program.

To set a breakpoint on an exception:

1. Choose Run | New Breakpoint (Ctrl-Shift-F8).
2. In the New Breakpoint dialog box, select Exception from the Breakpoint Type combo box.
3. In the Exception Class Name field, select the type of exception that you would like to set the breakpoint on.
4. In the Stop On combo box, select whether you want the breakpoint to apply to caught exceptions, uncaught exceptions, or both.

Setting a Breakpoint on a Field or Local Variable

You can set a breakpoint so that the debugger pauses when a field or variable is accessed (or only when the field or variable is modified).

To set a breakpoint on a field or variable:

1. Choose Run | New Breakpoint (Ctrl-Shift-F8).
2. In the New Breakpoint dialog box, select Variable from the Breakpoint Type combo box.
3. Fill in the Package Name, Class Name, and Field Name fields.
4. Select an option from the Stop On combo box.

If you select Variable Access, execution is suspended every time that field or variable is accessed in the code.

If you select Variable Modification, execution is only suspended if the field or variable is modified.

NetBeans IDE Tip

Most of the fields of the New Breakpoint dialog box are correctly filled in for you if you have the variable selected when you press Ctrl-Shift-F8. You might have to select (highlight) the whole variable name for this to work. Otherwise, information for the method that contains the variable might be filled in instead.

Setting a Breakpoint on the Start or Death of a Thread

You can monitor the creation or death of threads in your program by setting a breakpoint to have execution suspended every time a new thread is created or ended.

To set a breakpoint on threads:

1. Choose Run | New Breakpoint (Ctrl-Shift-F8).
2. In the New Breakpoint dialog box, select Thread from the Breakpoint Type combo box.
3. In the Set Breakpoint On field, select Thread Start, Thread Death, or Thread Start or Death.

Managing Breakpoints

You can use the Breakpoints window as shown in Figure 5-5 to manage breakpoints in one place. You can put breakpoints in groups, temporarily disable breakpoints, and provide customizations to the breakpoints from this window. To open the Breakpoints window, choose Window | Debugging | Breakpoints or press Alt-Shift-5.

Figure 5-5
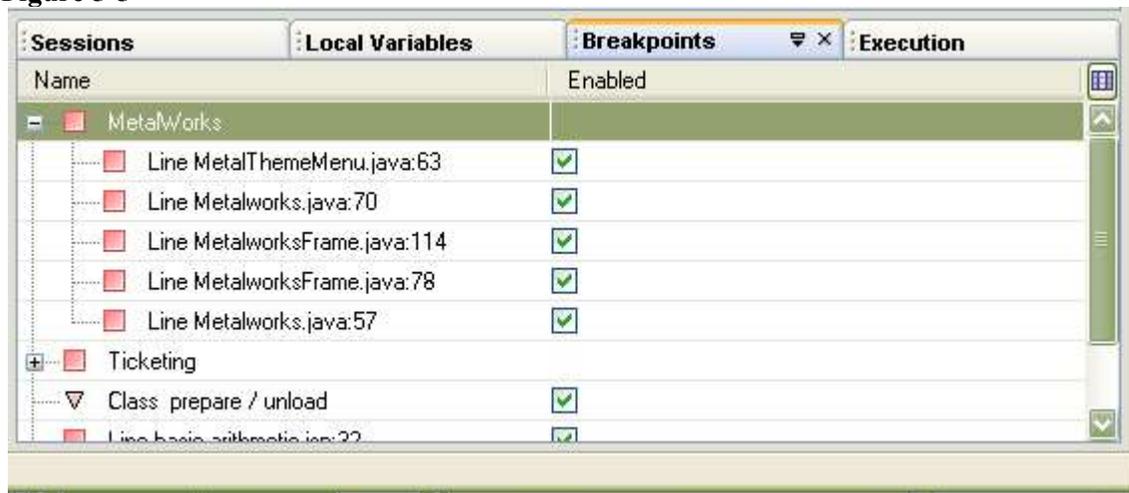
Breakpoints window

Grouping Related Breakpoints

In some cases, you might have a several related breakpoints that you would like to be able enable, disable, or delete together. Or maybe you merely want to consolidate some breakpoints under one node to make the Breakpoints window less cluttered.

To group some breakpoints:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).
2. Shift-click or Ctrl-click to select the breakpoints that you want to group. Then right-click the selection and choose Set Group Name.

The breakpoints are grouped under an expandable node.

Enabling and Disabling Breakpoints

You might find it useful to keep breakpoints set throughout your application, but you might not want to have all of the breakpoints active at all times. If this is the case, you can disable a breakpoint or breakpoint group and preserve it for later use.

To disable a breakpoint or breakpoint group:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (or by pressing Alt-Shift-5).
2. In the Breakpoints window, right-click the breakpoint or breakpoint group and choose disable.

Deleting a Breakpoint

To delete a line breakpoint, click the left margin of the line that has the breakpoint or click in the line and press Ctrl-F8.

To delete another type of breakpoint:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (or by pressing Alt-Shift-5).
2. In the Breakpoints window, right-click the breakpoint and choose Delete.

Customizing Breakpoint Behavior

There are a number of things that you can do to customize when a breakpoint is hit and what happens in the IDE when a breakpoint is hit. The following sub-topics cover some of those things.

Logging Breakpoints Without Suspending Execution

If you would like to monitor when a breakpoint is hit without suspending execution each time the breakpoint is hit, you can configure the breakpoint to not cause suspension of execution. When such a breakpoint is hit in the code, a message is printed in the Debugger Console window.

To turn off suspension of execution when a breakpoint is hit:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).

2. In the Breakpoints window, double-click the breakpoint to open the Customize Breakpoint window. (For line breakpoints, right-click the breakpoint and choose Customize.)

3. In the Action combo box, select No Thread (Continue).

Customizing Console Messages When Breakpoints Are Hit

You can customize the text that is printed to the console when a breakpoint is hit in your code.

To customize the console message that is printed when a breakpoint is reached:

1. Open the Breakpoints window by choosing Window | Debugging | Breakpoints (Alt-Shift-5).

2. In the Breakpoints window, double-click the breakpoint to open the Customize Breakpoint window. (For line breakpoints, right-click the breakpoint and choose Customize.)

3. In the Print Text combo box, modify the text that you want printed.

To make the printed text more meaningful, you can use some substitution codes to have things like the thread name and the line number printed. See Table 5-6 for a list of the substitution codes available.

Table 5-6: Substitution Codes for Breakpoint Console Text

Substitution Code	Prints
{className}	The name of the class where the breakpoint is hit. This code does not work for thread breakpoints.
{lineNumber}	The line number at which execution is suspended. This code does not work for thread breakpoints.
{methodName}	The method in which execution is suspended. This code does not work for thread breakpoints.
{threadName}	The thread in which the breakpoint is hit.
{variableValue}	The value of the variable (for breakpoints set on variables) or the value of the exception (for exception breakpoints).
{variableType}	The variable type (for breakpoints set on variables) or the exception type (for exception breakpoints).

Making Breakpoints Conditional

You can set up a breakpoint to only suspend execution of the code if a certain condition is met. For example, if you have a long For loop and you want to see what happens just before the loop finishes, you can make the breakpoint contingent on the iterator reaching a certain value.

Here are some examples of conditions you can place upon a breakpoint:

- `i==4` (which means that the execution would only stop on the breakpoint if the variable `i` equals 4 in the current scope)

- `ObjectVariable!=null` (which means that execution would not stop at the breakpoint until `ObjectVariable` is assigned a value)

- `MethodName` (where `Method` has a boolean return type and execution would stop at the breakpoint only if `Method` returned `true`.

- *CollectionX*.contains(*ObjectX*) (which means that execution would only stop at the breakpoint if *ObjectX* was in the collection.

To make a breakpoint conditional:

1. Open the Breakpoints window by pressing Alt-Shift-5.
2. In the Breakpoints window, right-click the breakpoint that you want to place a condition on and choose Customize.
3. In the Customize Breakpoint dialog box, fill in the Condition field with the condition that needs to be satisfied for execution to be suspended at the breakpoint.

Conditional breakpoints are marked with the ⊓ icon.

Monitoring Variables and Expressions

As you step through a program, you can monitor the running values of fields and local variables in the following ways:

- By holding the cursor over an identifier in the Source Editor to display a tooltip containing the value of the identifier in the current debugging context.
- By monitoring the values of variables and fields displayed in the Local Variables window.
- By setting a watch for an identifier or other expression and monitoring its value in the Watches window.

The Local Variables window (as shown in Figure 5-6) displays all variables that are currently in relevant in the current execution context of the program and provides and lists their types and values. If the value of the variable is an object reference, the value is given with the pound sign (#) and a number which serves as an identifier of the object's instance. You can jump to the source code for a variable by double-clicking the variable name.

Figure 5-6
*Local Variables window*
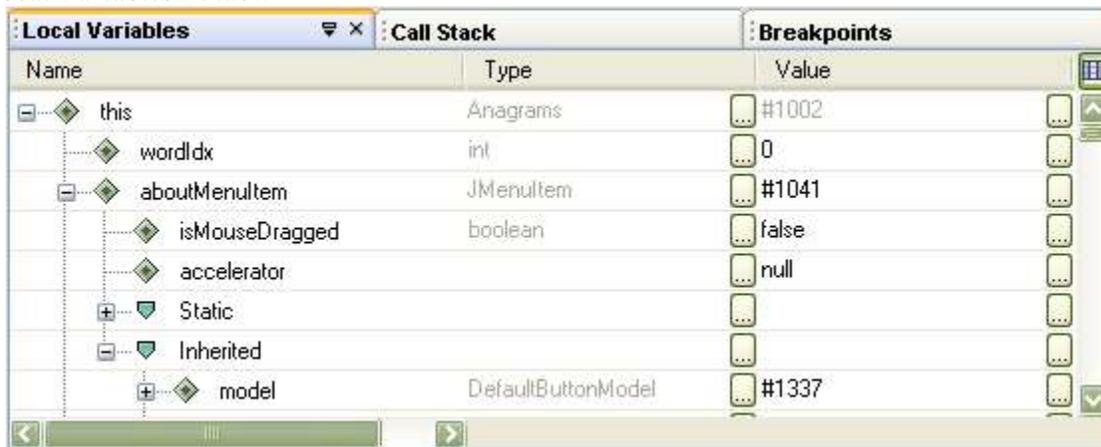
You can also create a more custom view of variables and expressions by setting watches and viewing them in the Watches window.

The Watches window (Alt-Shift-2) as shown in figure 5-7 is distinct from the Local Variables window in the following ways:

- The Watches window shows values for variables or expressions that you specify, which keeps the window uncluttered.
- The Watches window displays all watches that you have set, whether or not the variables are in context. If the variable exists separately in different contexts, the value given in the watches window applies to the value in the current context (not necessarily the context in which the watch was set).
- Watches persist across debugging sessions.

Figure 5-7
Watches window
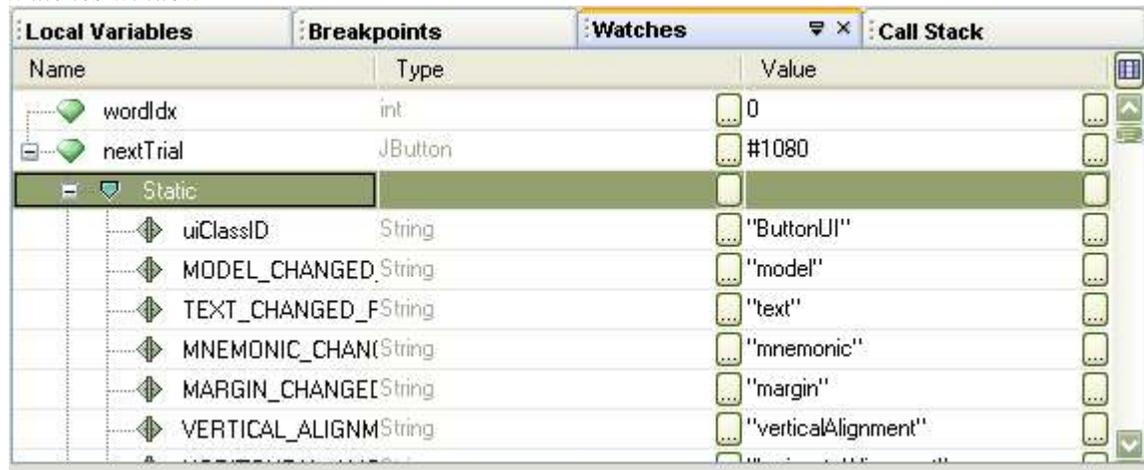
Setting a Watch on a Variable, Field, or Other Expression

To set a watch on an identifier such as a variable or field, right-click the variable or field in the Source Editor and choose New Watch. The identifier is then added to the Watches window.

To create a watch for another expression:

1. Choose Run | New Watch.
2. In the New Watch dialog box, type in an expression that you want evaluated.

   The expression must be written in Java syntax and can include local variables, fields of the current object, static fields, and method calls.

   You can even create an instance of an object and call one of its methods. When doing so, be sure to refer the class by its fully-qualified name.

Monitoring the Object Assigned to a Variable

You can create a so-called *fixed watch* to monitor an object that is assigned to a variable (rather than the value of the variable itself).

To create a fixed watch:

1. After starting a debugging session, open the Local Variables window (Alt-Shift-1).

2. Right-click the variable that you would like to set the fixed watch for and choose Create Fixed Watch.

A fixed watch is then added to the Watches window with an ◆ icon. Since a fixed watch applies to a specific object instance created during the debugging session, the fixed watch is removed when the debugging session is finished.

Displaying the Value of a Class's toString Method

You can add a column to the Local Variables and Watches windows to display the results of an object's `toString()` method. Doing so provides a way to get more useful information (such as the values of currently assigned fields) on an object than the numeric identifier of the object's instance that the Value column provides.

To display the toString() column in one of those windows:

1. Open the Local Variables window (Alt-Shift-1) or the Watches window (Alt-Shift-2).
2. Click the ▦ button in the upper-right corner of the window.
3. In the Change Visible Columns dialog box, select the `toString()` checkbox.

NetBeans IDE Tip
If you do not see the toString() column initially appear, try narrowing the width of the Value column to provide room for the toString() column to appear.

Changing Values of Variables or Expressions

As you debug a program, you can change the value of a variable or expression that is displayed in the Local Variables or Watches window. For example, you might increase the value of an iterator to get to the end of a loop faster.

To change the value of a variable:

1. Open the Watches window or Local Variables window.
2. In the Value field of the variable or expression, type in the new value and press Enter.

Displaying Variables From Previous Method Calls

The Call Stack window displays all of the calls within the current chain of method calls. If you would like to view the status of variables at another call in the chain, you can open the Call Stack window (Alt-Shift-3), right-click the method's node, and choose Make Current.

Making a different method current does not change the location of the program counter. If you continue execution with one of the step commands or the Continue command, the program will resume from where execution was suspended.

Backing up From a Method to its Call

Under some circumstances, it might be useful for you to step back in your code. For example, if you hit a breakpoint and would like to see how the code leading up to that breakpoint works, you can remove ("pop") the current call from the call stack to re-execute the method.

You can open the Call Stack window to view all of the method calls within the current chain of method calls in the current thread. The current call is marked with the ▣ icon. Other calls in the stack are marked with the ▢ icon.

To back up to a previous method call:

1. Open the Call Stack window (Alt-Shift-3).
2. Right-click the line in the Call Stack window that represents the place in the code that you want to return to and choose Pop to Here.

   The program counter returns to the line where the call was made. You can then re-execute the method.

NetBeans IDE Tip

When you pop a call, the effects of the previously executed code are not undone, so re-executing the code might cause the program to behave differently than it would during normal execution.

Monitoring and Controlling Execution of Threads

The IDE's Threads window (Alt-Shift-7) as shown in Figure 5-8 enables you to view the status of threads in the currently debugged program. It also enables you to change the thread that is being monitored in other debugger windows (such as Call Stack and Local Variables) and to suspend individual threads. See Table 5-7 for a guide to the icons used in the Threads window.

Changing the current thread does not affect the way the program executes.

Figure 5-8
Threads window

Table 5-7: Key to Icons in the Threads window

Icon	Meaning
	Currently monitored thread
	Currently monitored thread group
	Running thread
	Suspended thread
	Thread group

Switching the Currently Monitored Thread

The contents of the Call Stack and Local Variables windows are dependent on the thread being currently monitored in the debugger (otherwise known as the *current thread*). To switch the currently monitored thread:

1. Open the Threads window by pressing Alt-Shift-7.
2. Right-click the thread that you want to monitor and choose Make Current.

Suspending a Single Thread

By default, when your program hits a breakpoint, all threads are suspended. However, you can also configure a breakpoint so that only its thread is suspended when the breakpoint is hit:

1. Open the Breakpoints window by pressing Alt-Shift-5.
2. In the Breakpoints window, right-click the breakpoint and choose Customize.
3. In the Customize Breakpoint dialog box, select Current from the Suspend combo box.

Isolating Debugging to a Single Thread

By default, all threads in the application are executed in the debugger. If you would like to isolate the debugging so that only one thread is run in the debugger:

1. Make sure that the thread that you want debugged is designated as the current thread in the Threads window (Alt-Shift-7). The current thread is marked with the icon.
2. Open the Sessions window by pressing Alt-Shift-6.
3. In the Sessions window, right-click the session's node and choose Scope | Debug Current Thread.

Fixing Code During a Debugging Session

Using the IDE's Fix feature, it is possible to fine-tune code in the middle of a debugging session and continue debugging without starting a new debugging session. This can save you a lot of time that would otherwise be spent waiting for sources to be rebuilt and restarting your debugging session.

The Fix functionality is useful for situations such as when you need to:

• Fine-tune the appearance of a visual component that you have created.

- Change the logic within a method.

Fixing does not work if you do any of the following during the debugging session:

- Add or remove methods or fields.
- Change the access modifiers of a class, field, or method.
- Refactor the class hierarchy.
- Change code that has not yet been loaded into the virtual machine.

To use the fix command while debugging:

1. When execution is suspended during a debugging session, make whatever code changes are necessary in the Source Editor.
2. Choose Run | Fix to recompile the file and make the recompiled class available to the debugger.
3. Load the fixed code into the debugger.

   If you are changing the current method, this is done automatically. The method is automatically "popped" from the call stack, meaning that the program counter returns to the line where the method is called. You can then run the changed code by stepping back into the method (F7) or stepping over the method (F8).

   For a UI element, such as a JDialog component, you can close the component (or the component's container) and then re-open it to load the fixed code in the debugger.
4. Repeat steps 1-3 as necessary.

Viewing Multiple Debugger Windows Simultaneously

By default, the debugger's windows appear in a tabbed area in the lower right corner of the IDE in which only one of the tabs is viewable at a time. If you want to view multiple debugger windows simultaneously, you can use drag and drop to split a tab into it's own window or to move the tab into a different window (e.g. the window occupied by the Debugger Console). You can also drag the splitter between windows to change the size of each window.

To create a separate window area for a debugger tab, drag the window's tab to one side of the current window until a red outline for the location of the new window appears. Then release the mouse button. See Figures 5-9, 5-10, and 5-11 for snapshots of the process.

Figure 5-9
Call Stack window before moving it by dragging its tab to the left and dropping it in the lower left corner of the screen.
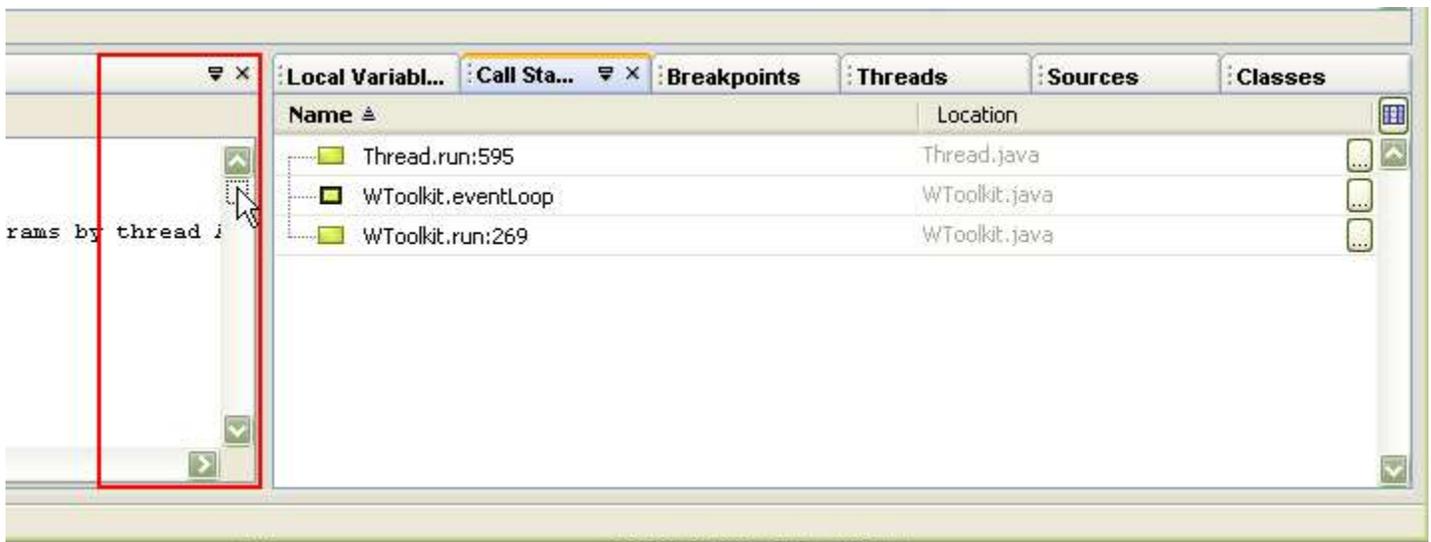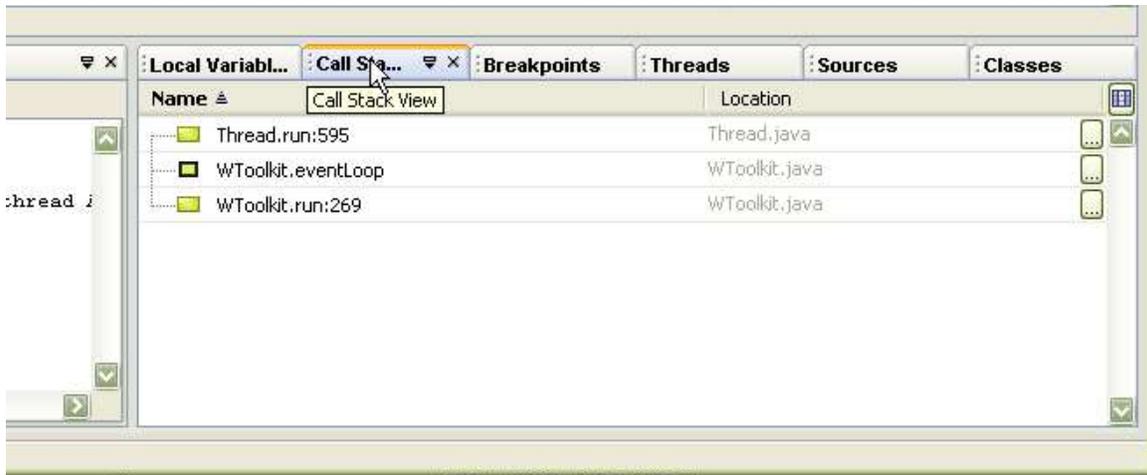
Figure 5-10
Call Stack window and and the outlined area to the left where it is to be dropped.

Figure 5-11
Call Stack after it has been moved to the new window area.

Call Stack	⤳ ✕
Name ≜	Loca...
🟩 Thre Thread.	...
⬛ WTo WToolki	...
🟩 WTo WToolki	...

Local Variables	Breakpoi... ⤳ ✕	Threads	Sources	Classes

Name	Enabled
⊞ 🟥 AnagramsBreakpoints	
⊞ 🟥 iTERATORS-bPS	
▽ Class PrintData prepare / unload	☐
🟥 Line Anagrams.java:353	☑
🟥 Line Iterators2.java:35	☐
🟥 Line Iterators2.java:36	☐
🟥 Line Iterators2.java:39	☐
▽ Method Anagrams.initComponents	☐
▽ Method ArrayList.add	☐