



# Building Applications with JBuilder®

---



VERSION 8

**Borland®**  
**JBuilder®**

Borland Software Corporation  
100 Enterprise Way, Scotts Valley, CA 95066-3249  
[www.borland.com](http://www.borland.com)

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1997–2002 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation in the United States and other countries. All other marks are the property of their respective owners.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JBE0080WW21001bajb 7E10R1002  
0203040506-9 8 7 6 5 4 3 2 1  
PDF

# Contents

## Chapter 1

### **Introduction 1-1**

Documentation conventions . . . . .	1-4
Developer support and resources . . . . .	1-6
Contacting Borland Technical Support. . . . .	1-6
Online resources . . . . .	1-6
World Wide Web . . . . .	1-6
Borland newsgroups . . . . .	1-7
Usenet newsgroups . . . . .	1-7
Reporting bugs . . . . .	1-7

## Chapter 2

### **Creating and managing projects 2-1**

Creating a new project. . . . .	2-2
Creating a new project with the Project wizard . . . . .	2-2
Selecting project name and template . . . . .	2-2
Setting project paths . . . . .	2-4
Setting general project settings . . . . .	2-4
Creating a project from existing files . . . . .	2-7
Selecting the source directory and name for your new JBuilder project . . . . .	2-8
Displaying files. . . . .	2-9
Switching between files . . . . .	2-9
Saving projects . . . . .	2-10
Opening an existing project. . . . .	2-10
Creating a new Java source file. . . . .	2-11
Managing projects . . . . .	2-13
Adding to a project. . . . .	2-13
Adding folders . . . . .	2-13
Adding files and packages. . . . .	2-14
Removing material from a project . . . . .	2-14
Deleting material. . . . .	2-15
Opening a file outside of a project . . . . .	2-15
Renaming projects and files. . . . .	2-15
Adding a new directory view. . . . .	2-16
Setting project properties . . . . .	2-17
Setting the JDK . . . . .	2-19
Editing the JDK. . . . .	2-19
Debugging with -classic . . . . .	2-20
Setting the JDK in SE and Enterprise . . . . .	2-20
Configuring JDKs . . . . .	2-22
Setting paths for required libraries . . . . .	2-22

Working with multiple projects . . . . .	2-23
Switching between projects . . . . .	2-23
Saving multiple projects . . . . .	2-24
More information about projects . . . . .	2-24

## Chapter 3

### **Working with project groups 3-1**

Creating project groups . . . . .	3-1
Adding and removing projects from project groups . . . . .	3-3
Navigating project groups . . . . .	3-4
Adding projects as required libraries . . . . .	3-4

## Chapter 4

### **Managing paths 4-1**

Working with libraries . . . . .	4-1
Adding and configuring libraries . . . . .	4-2
Editing libraries . . . . .	4-5
Adding projects as required libraries . . . . .	4-5
Display of library lists . . . . .	4-6
Packages . . . . .	4-6
java file location = source path + package path . . . . .	4-7
.class file location = output path + package path . . . . .	4-8
Using packages in JBuilder. . . . .	4-8
Package naming guidelines . . . . .	4-9
How JBuilder constructs paths . . . . .	4-9
Source path. . . . .	4-10
Output path . . . . .	4-10
Class path . . . . .	4-10
Browse path . . . . .	4-11
Doc path . . . . .	4-11
Backup path . . . . .	4-12
Working directory. . . . .	4-12
Where are my files? . . . . .	4-12
How JBuilder finds files when you drill down . . . . .	4-13
How JBuilder finds files when you compile . . . . .	4-13
How JBuilder finds class files when you run or debug . . . . .	4-13

## Chapter 5

### Compiling Java programs 5-1

Smart dependencies checking . . . . .	5-2
Compiling a program . . . . .	5-3
JBuilder build menus . . . . .	5-3
Building projects with the Run command . . . . .	5-4
Syntax errors and error messages . . . . .	5-4
Compile problems when opening projects. . . . .	5-5
Checking for package/directory correspondence . . . . .	5-6
Setting compiler options . . . . .	5-6
Specifying a compiler . . . . .	5-8
Setting additional compiler and build options. . . . .	5-8
Setting the output path . . . . .	5-8
Compiling projects within a project group . . . . .	5-9
Compiling from the command line . . . . .	5-9
bmj (Borland Make for Java) . . . . .	5-10
bcj (Borland Compiler for Java). . . . .	5-10
Building a project from the command line . . . . .	5-10
Switching between the command line and IDE . . . . .	5-11

## Chapter 6

### Building Java programs 6-1

The JBuilder build system. . . . .	6-1
Build system terms. . . . .	6-2
Build phases . . . . .	6-2
The Make command . . . . .	6-3
The Rebuild command . . . . .	6-4
The Clean command . . . . .	6-5
Building project groups . . . . .	6-5
Specifying the build order for a project group . . . . .	6-6
Building a project group. . . . .	6-6
Adding project group build targets to the Project menu . . . . .	6-7
Building with external Ant files . . . . .	6-7
Adding Ant build files to projects . . . . .	6-8
Adding Ant files with the Ant wizard . . . . .	6-9
Adding Ant files manually . . . . .	6-9
Creating and editing Ant build files . . . . .	6-10

Importing existing Ant projects . . . . .	6-10
Building Ant projects . . . . .	6-11
Specifying the JDK. . . . .	6-12
Building Ant projects with the Run command . . . . .	6-12
Setting Ant properties. . . . .	6-12
Ant options. . . . .	6-14
Adding custom Ant libraries. . . . .	6-14
Building SQLJ files . . . . .	6-15
Creating external build tasks . . . . .	6-16
External Build Task wizard. . . . .	6-16
Building external tasks . . . . .	6-17
Setting external build task properties. . . . .	6-17
Configuring the Project menu . . . . .	6-18
Configuring the Project menu for project groups . . . . .	6-19
Automatic source packages . . . . .	6-21
Filtering packages. . . . .	6-23
Excluding packages . . . . .	6-24
Including packages . . . . .	6-25
Selective resource copying. . . . .	6-25
Individual resource properties. . . . .	6-25
File-specific options . . . . .	6-26
Project-wide options. . . . .	6-27
Adding unrecognized file types as generic resource files. . . . .	6-27
Project Properties Resource page . . . . .	6-28

## Chapter 7

### Running Java programs 7-1

Running program files . . . . .	7-1
Running web files. . . . .	7-2
Running projects . . . . .	7-3
Using the Run command . . . . .	7-3
Running grouped projects . . . . .	7-5
Running OpenTools. . . . .	7-5
Setting runtime configurations . . . . .	7-6
Creating a runtime configuration . . . . .	7-8
Editing a runtime configuration . . . . .	7-10
Build Targets. . . . .	7-10
Runtime configuration types. . . . .	7-12
Running programs from the command line . . . . .	7-13
Running a deployed program from the command line . . . . .	7-13

## Chapter 8

### Debugging Java programs 8-1

Types of errors . . . . .	8-2
Runtime errors . . . . .	8-2
Logic errors . . . . .	8-2
Overview of the debugging process . . . . .	8-3
Creating a runtime configuration. . . . .	8-3
Compiling the project with symbolic debug information . . . . .	8-4
Starting the debugger . . . . .	8-6
Starting the debugger with the -classic option . . . . .	8-7
Running under the debugger's control . . . . .	8-7
Pausing program execution . . . . .	8-8
Ending a debugging session . . . . .	8-8
The debugger user interface . . . . .	8-8
Debugging sessions . . . . .	8-9
Debugger views . . . . .	8-10
Console output, input, and errors view . . . . .	8-11
Classes with tracing disabled view . . . . .	8-12
Data and code breakpoints view . . . . .	8-13
Threads, call stacks, and data view . . . . .	8-15
Data watches view . . . . .	8-18
Loaded classes and static data view . . . . .	8-20
Synchronization monitors view . . . . .	8-22
Debugger toolbar. . . . .	8-23
Debugger shortcut keys . . . . .	8-25
ExpressionInsight . . . . .	8-25
Tool tips . . . . .	8-26
Debugging non-Java source . . . . .	8-27
Controlling program execution. . . . .	8-27
Running and suspending your program . . . . .	8-27
Resetting the program . . . . .	8-28
The execution point . . . . .	8-28
Setting the execution point . . . . .	8-29
Managing threads . . . . .	8-30
Using the split pane . . . . .	8-30
Displaying only the current thread . . . . .	8-31
Displaying the top stack frame . . . . .	8-31
Choosing the thread to step into . . . . .	8-31
Keeping a thread suspended . . . . .	8-31
Detecting deadlock states . . . . .	8-32
Moving through code . . . . .	8-32
Stepping into a method call . . . . .	8-33
Stepping over a method call . . . . .	8-33
Stepping out of a method . . . . .	8-34
Using Smart Step. . . . .	8-34
Running to a breakpoint . . . . .	8-35
Running to the end of a method . . . . .	8-36
Running to the cursor location. . . . .	8-36
Viewing method calls . . . . .	8-36
Locating a method call . . . . .	8-37
Controlling which classes to trace into . . . . .	8-37
Tracing into classes with no source available . . . . .	8-39
Breakpoints and tracing disabled settings . . . . .	8-40
Using breakpoints. . . . .	8-40
Setting breakpoints . . . . .	8-41
Setting a line breakpoint . . . . .	8-41
Setting an exception breakpoint . . . . .	8-43
Setting a class breakpoint . . . . .	8-45
Setting a method breakpoint . . . . .	8-46
Setting a field breakpoint . . . . .	8-47
Setting a cross-process breakpoint. . . . .	8-47
Setting breakpoint properties . . . . .	8-50
Setting breakpoint actions . . . . .	8-51
Stopping program execution . . . . .	8-51
Logging a message. . . . .	8-51
Creating conditional breakpoints . . . . .	8-52
Setting the breakpoint condition . . . . .	8-53
Using pass count breakpoints . . . . .	8-53
Disabling and enabling breakpoints. . . . .	8-54
Deleting breakpoints . . . . .	8-54
Locating line breakpoints. . . . .	8-55
Examining program data values . . . . .	8-55
How variables are displayed in the debugger . . . . .	8-56
Changing data values. . . . .	8-57
Watching expressions . . . . .	8-59
Variable watches . . . . .	8-59
Object watches . . . . .	8-61
Editing a watch. . . . .	8-61
Deleting a watch . . . . .	8-62
Evaluating and modifying expressions . . . . .	8-62
Evaluating expressions . . . . .	8-62
Evaluating method calls. . . . .	8-62
Modifying the values of variables . . . . .	8-63
Modifying code while debugging. . . . .	8-64
Updating all class files . . . . .	8-64
Updating individual class files. . . . .	8-65

Resetting the execution point . . . . .	8-65
Options for modifying code. . . . .	8-65
Customizing the debugger . . . . .	8-67
Customizing the debugger display. . . . .	8-67
Setting debug configuration options. . . . .	8-68
Setting update intervals . . . . .	8-69

## Chapter 9

### Remote debugging 9-1

Launching and debugging a program	
on a remote computer . . . . .	9-2
Debugging a program already running	
on the remote computer . . . . .	9-6
Debugging local code running in a	
separate process . . . . .	9-9
Debugging with cross-process breakpoints . . . . .	9-10

## Chapter 10

### Creating JavaBeans with BeansExpress 10-1

What is a JavaBean? . . . . .	10-1
Why build JavaBeans?. . . . .	10-1
Generating a bean class . . . . .	10-2
Designing the user interface of your bean . . . . .	10-4
Adding properties to your bean . . . . .	10-4
Modifying a property . . . . .	10-6
Removing a property . . . . .	10-7
Adding bound and constrained	
properties . . . . .	10-7
Creating a BeanInfo class . . . . .	10-8
Specifying BeanInfo data for a property. . . . .	10-9
Working with the BeanInfo designer. . . . .	10-9
Modifying a BeanInfo class . . . . .	10-10
Adding events to your bean . . . . .	10-11
Firing events . . . . .	10-11
Listening for events . . . . .	10-14
Creating a custom event set. . . . .	10-15
Creating a property editor . . . . .	10-17
Creating a String List editor. . . . .	10-17
Creating a String Tag List editor . . . . .	10-18
Creating an Integer Tag List editor . . . . .	10-19
Creating a custom component	
property editor . . . . .	10-20
Adding support for serialization. . . . .	10-21
Checking the validity of a JavaBean . . . . .	10-21
Installing a bean on the component	
palette . . . . .	10-22

## Chapter 11

### Visualizing code with UML 11-1

Java and UML . . . . .	11-2
Java and UML terms . . . . .	11-2
JBuilder and UML . . . . .	11-3
Limited package dependency diagram . . . . .	11-4
Combined class diagram . . . . .	11-4
JBuilder UML diagrams defined. . . . .	11-7
Visibility icons . . . . .	11-9
Viewing UML diagrams . . . . .	11-10
JBuilder's UML browser . . . . .	11-11
Viewing package diagrams. . . . .	11-12
Viewing class diagrams. . . . .	11-12
Viewing inner classes . . . . .	11-12
Viewing source code . . . . .	11-13
Viewing Javadoc. . . . .	11-13
Using the context menu. . . . .	11-14
Scrolling the view . . . . .	11-14
Full view . . . . .	11-15
Partial view. . . . .	11-15
Refreshing the view . . . . .	11-15
Navigating diagrams . . . . .	11-15
UML and the structure pane . . . . .	11-16
Package diagrams . . . . .	11-16
Class diagrams. . . . .	11-17
Customizing UML diagrams . . . . .	11-17
Setting project properties. . . . .	11-17
Filtering packages and classes . . . . .	11-18
Including references from project	
libraries . . . . .	11-18
Including references from	
generated source . . . . .	11-19
Setting IDE Options. . . . .	11-19
Creating images of UML diagrams . . . . .	11-20
Printing UML diagrams . . . . .	11-20
Refactoring and Find References . . . . .	11-21

## Chapter 12

### Refactoring code symbols 12-1

Types of refactorings . . . . .	12-1
Optimize Imports . . . . .	12-2
Rename refactoring . . . . .	12-2
Move refactoring . . . . .	12-3
Change Parameters . . . . .	12-4
Extract Method . . . . .	12-4
Introduce Variable. . . . .	12-4
Surround With Try/Catch . . . . .	12-4

JBuilder's refactoring tools . . . . .	12-5
Setting up for references discovery and refactoring . . . . .	12-6
Learning about a symbol before refactoring . . . . .	12-7
Finding a symbol's definition . . . . .	12-7
Finding references to a symbol . . . . .	12-8
Viewing changes before a refactoring . . . . .	12-10
Executing a refactoring . . . . .	12-13
Optimizing imports . . . . .	12-14
Using Optimize Imports . . . . .	12-16
Rename refactoring a package . . . . .	12-17
Rename refactoring a class . . . . .	12-17
Move refactoring a class . . . . .	12-18
Rename refactoring a method . . . . .	12-19
Rename refactoring a local variable . . . . .	12-20
Rename refactoring a field . . . . .	12-21
Rename refactoring a property . . . . .	12-22
Changing method parameters . . . . .	12-22
Extracting a method . . . . .	12-24
Introducing a variable . . . . .	12-25
Surrounding a block with try/catch . . . . .	12-26
Undoing a refactoring . . . . .	12-26
Saving refactorings . . . . .	12-26

## Chapter 13

### Unit testing 13-1

JUnit . . . . .	13-1
Cactus . . . . .	13-2
Unit testing features in JBuilder . . . . .	13-2
Discovering tests . . . . .	13-3
JUnit Test Collector . . . . .	13-3
Creating JUnit test cases and test suites . . . . .	13-4
The Test Case wizard . . . . .	13-5
Adding test code to your test cases . . . . .	13-6
The Test Suite wizard . . . . .	13-7
The EJB Test Client wizard . . . . .	13-7
Using predefined test fixtures . . . . .	13-8
JDBC fixture . . . . .	13-8
JNDI fixture . . . . .	13-9
Comparison fixture . . . . .	13-10
Creating a custom test fixture . . . . .	13-11
Working with Cactus . . . . .	13-11
Cactus Setup wizard . . . . .	13-11
Creating a Cactus test case for your Enterprise JavaBean . . . . .	13-12
Running Cactus tests . . . . .	13-13
Running tests . . . . .	13-13
JBTestRunner . . . . .	13-14

Test Hierarchy . . . . .	13-15
Test Failures . . . . .	13-15
Test Output . . . . .	13-15
JUnit TextUI . . . . .	13-16
JUnit SwingUI . . . . .	13-16
Runtime configurations . . . . .	13-16
Defining a test stack trace filter . . . . .	13-16
Debugging tests . . . . .	13-17

## Chapter 14

### Creating Javadoc from API source files 14-1

Adding Javadoc comments to your API source files . . . . .	14-2
Where to place Javadoc comments . . . . .	14-3
Javadoc tags . . . . .	14-5
Automatically generating Javadoc tags . . . . .	14-6
Javadoc @todo tags . . . . .	14-7
Conflicts in Javadoc comments . . . . .	14-8
Generating the documentation node . . . . .	14-8
Choosing the format of the documentation . . . . .	14-9
Choosing documentation build options . . . . .	14-10
Choosing the packages to document . . . . .	14-12
Specifying doclet command-line options . . . . .	14-13
Generating the output files . . . . .	14-16
Generating additional files . . . . .	14-18
Package-level files . . . . .	14-18
Overview comment files . . . . .	14-20
Viewing Javadoc . . . . .	14-20
How JBuilder displays Javadoc . . . . .	14-22
Maintaining Javadoc . . . . .	14-22
Changing properties for the documentation node . . . . .	14-23
Changing node properties . . . . .	14-23
Changing Javadoc properties . . . . .	14-24
Changing doclet properties . . . . .	14-24
Creating a documentation archive file . . . . .	14-25
Creating a custom doclet . . . . .	14-27

## Chapter 15

### Deploying Java programs 15-1

Deploying to Java archive files (JAR) . . . . .	15-2
Understanding the manifest file . . . . .	15-3
Deployment strategies . . . . .	15-4
Using the JDK Java Archive Tool . . . . .	15-5
Running a program from a JAR file . . . . .	15-5

Viewing archive file contents . . . . .	15-6
Updating the contents of a JAR file. . . . .	15-7
Deployment issues. . . . .	15-7
Is everything you need on the class path? . . . . .	15-8
Does your program rely on JDK 1.1.x or Java 2 (JDK 1.2 and above) features? . . . .	15-8
Does the user already have Java libraries installed locally? . . . . .	15-9
Is this an applet or an application? . . . .	15-9
Download time . . . . .	15-10
Deployment quicksteps . . . . .	15-10
Applications . . . . .	15-11
Applets . . . . .	15-11
JavaBeans . . . . .	15-13
Deployment tips . . . . .	15-14
Setting up your working environment. . . .	15-14
Internet deployment . . . . .	15-14
Deploying distributed applications . . . .	15-15
Redistribution of classes supplied with JBuilder . . . . .	15-15
Additional deployment information. . . . .	15-16
Deploying with the Archive Builder. . . . .	15-17
The Archive Builder and resources. . . . .	15-17
Selecting an archive type . . . . .	15-17
Specifying the file to be created. . . . .	15-18
Choosing deployment descriptor files . . .	15-20
Specifying the parts of the project to archive . . . . .	15-21
Specifying archive content for a Resource Adapter archive . . . . .	15-22
Determining library dependencies . . . . .	15-23
Setting archive manifest options . . . . .	15-24
Selecting a method for determining the application's main class . . . . .	15-25
Determining which executable files to build. . . . .	15-26
Running executables . . . . .	15-27
Setting runtime configuration options . . .	15-28
Creating executables with the Native Executable Builder . . . . .	15-29
Generating archive files . . . . .	15-31
Understanding archive nodes . . . . .	15-31
Viewing the archive and manifest . . . .	15-31
Modifying archive node properties . . . .	15-32
Removing, deleting, and renaming archives . . . . .	15-33

<b>Chapter 16</b>	
<b>Internationalizing programs with JBuilder</b>	<b>16-1</b>
Internationalization terms and definitions. . .	16-1
Internationalization features in JBuilder . . .	16-2
A multilingual sample application . . . . .	16-3
Eliminating hard-coded strings . . . . .	16-5
Using the Resource Strings wizard . . . . .	16-5
Using the Localizable Property Setting dialog box . . . . .	16-8
dbSwing internationalization features . . . . .	16-8
Using JBuilder's locale-sensitive components . . . . .	16-9
JBuilder components display any Unicode character . . . . .	16-10
Internationalization features in the UI designer . . . . .	16-10
Unicode in the IDE debugger . . . . .	16-12
Specifying a native encoding for the compiler . . . . .	16-12
Setting the encoding option . . . . .	16-12
Native encodings supported . . . . .	16-13
Adding and overriding encodings . . . . .	16-13
More about native encodings . . . . .	16-14
The 16-bit Unicode format . . . . .	16-14
Unicode support using ASCII and '\u' . . .	16-15
JBuilder around the world . . . . .	16-15
Online internationalization support . . . . .	16-16

<b>Chapter 17</b>	
<b>Tutorial: Compiling, running, and debugging</b>	<b>17-1</b>
Step 1: Opening the sample project . . . . .	17-2
Step 2: Fixing syntax errors . . . . .	17-3
Step 3: Fixing compiler errors . . . . .	17-4
Step 4: Running the program . . . . .	17-7
Saving files and running the program . . .	17-9
Step 5: Fixing the subtractValues() method . . . . .	17-10
Saving files and running the program . . .	17-15
Step 6: Fixing the divideValues() method . . .	17-16
Saving files and running the program . . .	17-19
Step 7: Fixing the oddEven() method . . . . .	17-19
Step 8: Finding runtime exceptions . . . . .	17-23



<b>Chapter 18</b>	
<b>Tutorial: Building with Ant files</b>	<b>18-1</b>
Step 1: Creating a project and application. . . . .	18-2
Step 2: Creating the Ant build file . . . . .	18-2
Step 3: Executing individual targets . . . . .	18-3
Step 4: Executing the default target . . . . .	18-4
Step 5: Handling errors with Ant . . . . .	18-5
Step 6: Adding a target to the Project menu. . . . .	18-6
Step 7: Setting Ant properties. . . . .	18-7
Step 8: Adding custom Ant tasks to your project . . . . .	18-9

<b>Chapter 19</b>	
<b>Tutorial: Remote debugging</b>	<b>19-1</b>
Step 1: Opening the sample project . . . . .	19-2
Step 2: Setting runtime and debugging configurations . . . . .	19-3
Step 3: Setting breakpoints . . . . .	19-7
Step 4: Compiling the server and copying server class files to the remote computer . . . . .	19-9
Step 5: Starting the RMI Registry and server on the remote computer . . . . .	19-10
Step 6: Starting the server process and the client in debug mode and stepping into the cross-process breakpoint . . . . .	19-11

<b>Chapter 20</b>	
<b>Tutorial: Visualizing code with the UML browser</b>	<b>20-1</b>
Step 1: Compiling the sample . . . . .	20-2
Step 2: Viewing a UML package diagram. . . . .	20-3
Step 3: Viewing a UML class diagram . . . . .	20-5
Step 4: Adding references from libraries . . . . .	20-9
Step 5: Filtering UML diagrams . . . . .	20-12

<b>Chapter 21</b>	
<b>Tutorial: Creating and running test cases and test suites</b>	<b>21-1</b>
Step 1: Opening an existing project . . . . .	21-2
Step 2: Creating skeleton test cases . . . . .	21-2
Step 3: Implementing a test method that throws an expected exception . . . . .	21-3
Viewing the test failure output . . . . .	21-4
Fixing the test so it passes . . . . .	21-4
Step 4: Writing a second test method . . . . .	21-5
Step 5: Creating a test suite . . . . .	21-5
Step 6: Running tests . . . . .	21-7

<b>Chapter 22</b>	
<b>Tutorial: Working with test fixtures</b>	<b>22-1</b>
Step 1: Creating a new project . . . . .	22-2
Step 2: Creating a Data Module . . . . .	22-2
Step 3: Creating a comparison fixture . . . . .	22-3
Step 4: Creating a JDBC fixture . . . . .	22-4
Step 5: Modifying the JDBC Fixture to run SQL scripts . . . . .	22-5
Step 6: Creating a test case using test fixtures . . . . .	22-6
Step 7: Implementing the test case . . . . .	22-7
Step 8: Adding a required library . . . . .	22-8
Step 9: Running the test case. . . . .	22-8

<b>Appendix A</b>	
<b>Creating configuration files for native executables</b>	<b>A-1</b>
Starting the VM . . . . .	A-3
Configuration file requirements. . . . .	A-3
File type and location . . . . .	A-3
Blank lines and comments . . . . .	A-3
Path conventions . . . . .	A-3
Directives . . . . .	A-4
javapath . . . . .	A-4
mainclass. . . . .	A-4
addpath. . . . .	A-4
addjars . . . . .	A-5
addbootpath . . . . .	A-5
addbootjars. . . . .	A-5
addskipppath . . . . .	A-6
vmparam. . . . .	A-6
include . . . . .	A-6
includedir . . . . .	A-6
copyenv. . . . .	A-7
exportenv. . . . .	A-7
addparam . . . . .	A-7
clearparams . . . . .	A-7
restartcode . . . . .	A-7
Optional all-in-one launcher support. . . . .	A-8

<b>Appendix B</b>	
<b>Using the command-line tools</b>	<b>B-1</b>
Setting the class path for command-line tools. . . . .	B-2
Using the -classpath option . . . . .	B-2

Setting the CLASSPATH environment		
variable for command-line tools . . . . .	B-2	
UNIX: CLASSPATH environment		
variable. . . . .	B-3	
Windows: CLASSPATH environment		
variable. . . . .	B-3	
JBuilder command-line interface. . . . .	B-4	
Accessing a list of options. . . . .	B-4	
Syntax . . . . .	B-5	
Options . . . . .	B-5	
Borland Compiler for Java (bcj) . . . . .	B-7	
Syntax . . . . .	B-7	
Description . . . . .	B-7	
Options . . . . .	B-8	
Cross-compilation options . . . . .	B-11	
VM options. . . . .	B-11	
Borland Make for Java (bmj). . . . .	B-12	
Syntax. . . . .	B-12	
Description. . . . .	B-12	
Options . . . . .	B-13	
Cross-compilation options . . . . .	B-16	
Specifiers for root classes . . . . .	B-17	
VM options. . . . .	B-18	
<b>Index</b>		<b>I-1</b>

# Tables

1.1	Typeface and symbol conventions . . . . .	1-4	8.16	Context menu with no selection in Data watches view . . . . .	8-20
1.2	Platform conventions . . . . .	1-5	8.17	Icons in Loaded classes and static data view . . . . .	8-21
4.1	Colors in library lists . . . . .	4-6	8.18	Context menu with selection in Loaded classes and static data view . . . .	8-21
6.1	Build system terms . . . . .	6-2	8.19	Context menu with no selection in Loaded classes and static data view . . . .	8-22
6.2	Build system phases . . . . .	6-3	8.20	Icons in Synchronization monitors view . . . . .	8-23
6.3	Package filtering icons . . . . .	6-25	8.21	Context menu in Synchronization monitors view . . . . .	8-23
8.1	Menu commands to start debugger . . . . .	8-6	8.22	Toolbar buttons . . . . .	8-24
8.2	Debugger views . . . . .	8-10	8.23	Debugger shortcut keys . . . . .	8-25
8.3	Icons in Console view . . . . .	8-11	8.24	Debugger features . . . . .	8-55
8.4	Context menu in Console view . . . . .	8-12	8.25	Types of scoped variable watches . . . . .	8-61
8.5	Icons in the Classes with tracing disabled view . . . . .	8-12	11.1	Java and UML terms . . . . .	11-2
8.6	Context menu with class/package selected in Classes with tracing disabled view . . . . .	8-12	11.2	UML diagram definitions . . . . .	11-7
8.8	Icons in Data and code breakpoints view . . . . .	8-13	11.3	UML visibility icons . . . . .	11-9
8.9	Context menu with breakpoint selected in Data and code breakpoints view . . . . .	8-13	12.1	Refactoring and code symbols . . . . .	12-3
8.7	Context menu with no selection in Classes with tracing disabled view . . . .	8-13	12.2	Find References details . . . . .	12-9
8.10	Context menu with no selection in Data and code breakpoints view . . . . .	8-14	12.3	Refactoring details . . . . .	12-11
8.11	Icons in Threads, call stacks, and data view . . . . .	8-15	14.1	Javadoc tags . . . . .	14-5
8.12	Context menu with selection in Threads, call stacks, and data view . . . .	8-16	14.2	Options not set in the wizard . . . . .	14-15
8.14	Icons in Data watches view . . . . .	8-18	19.1	Dialog box pages for setting client and server runtime and debugging configurations . . . . .	19-3
8.15	Context menu with watch selected in Data watches view . . . . .	8-18	19.2	Command line RMI and debugger arguments . . . . .	19-11
8.13	Context menu with no selection in Threads, call stacks, and data view . . . .	8-18	19.1	RMI client/server error messages . . . .	19-13

# Figures

6.1	Properties Resource page . . . . .	6-26	11.5	JBuilder's visibility icons . . . . .	11-9
7.1	Error messages in the AppBrowser. . . . .	7-4	11.6	UML browser. . . . .	11-11
8.1	The debugger user interface. . . . .	8-9	11.7	Viewing inner classes . . . . .	11-13
8.2	Debugger toolbar . . . . .	8-23	11.8	Structure pane for UML diagrams. . . . .	11-16
8.3	ExpressionInsight window . . . . .	8-26	12.1	Class references in the Search Results tab . . . . .	12-9
8.4	Tool tip window. . . . .	8-26	12.2	Method references in the Search Results tab . . . . .	12-10
8.5	The execution point. . . . .	8-29	12.3	Field and local variable references in the Search Results tab . . . . .	12-10
8.6	Threads, call stacks, and data view split pane . . . . .	8-31	12.4	Rename Class dialog box . . . . .	12-10
8.7	Synchronization monitors view . . . . .	8-32	12.5	Refactoring tab before refactoring . . . . .	12-11
8.8	Stub source file . . . . .	8-39	12.6	Refactoring tab after refactoring . . . . .	12-12
8.9	Stopped In Class With Tracing Disabled dialog box . . . . .	8-40	12.7	Source file and Refactoring tab after refactoring . . . . .	12-13
8.10	Data and code breakpoints view . . . . .	8-41	14.1	ToDo folder in structure pane . . . . .	14-7
8.11	Breakpoint actions . . . . .	8-51	14.2	Javadoc conflicts in structure pane. . . . .	14-8
8.12	Breakpoint status bar message . . . . .	8-51	14.3	Choose a doclet page . . . . .	14-9
8.13	Conditional breakpoints . . . . .	8-53	14.4	Specify project and build options page . . . . .	14-10
8.14	Loaded classes and static data view . . . . .	8-56	14.5	Select packages and visibility level page . . . . .	14-12
8.15	Threads, call stacks, and data view . . . . .	8-57	14.6	Specify doclet command-line options page . . . . .	14-13
8.16	Data watches view . . . . .	8-59	14.7	Documentation node in project pane . . . . .	14-16
8.17	Expression evaluation in the Evaluate/Modify dialog box . . . . .	8-62	14.8	Expanded documentation nodes. . . . .	14-20
8.18	Method evaluation in the Evaluate/Modify dialog box . . . . .	8-63	14.9	Index file output from Standard Doclet . . . . .	14-21
8.19	Debug page of Runtime Configuration Properties dialog box . . . . .	8-66	14.10	Index file output from JDK 1.1 Doclet . . . . .	14-21
11.1	Package diagram . . . . .	11-4	14.11	On-the-fly Javadoc output . . . . .	14-22
11.2	Class diagram . . . . .	11-5			
11.3	Class diagram with properties displayed separately . . . . .	11-6			
11.4	Class diagram without properties displayed separately . . . . .	11-6			



# Tutorials

Compiling, running, and debugging . . . . .	17-1
Building with Ant files . . . . .	18-1
Remote debugging. . . . .	19-1
Visualizing code with the UML browser . . . .	20-1
Creating and running test cases and test suites . . . . .	21-1
Working with test fixtures . . . . .	22-1



## Introduction

*Building Applications with JBuilder* explains how to use JBuilder's IDE to manage your projects and to compile, run, and debug your Java programs. It explains how to use BeansExpress to create JavaBeans. It also describes advanced techniques, such as deploying and internationalizing applications for different locales, visualizing code, refactoring and unit testing.

*Building Applications with JBuilder* contains the following chapters:

- [Chapter 2, "Creating and managing projects"](#)  
Explains how to work with JBuilder projects and set project properties.
- [Chapter 3, "Working with project groups"](#)  
Describes how to place related projects in project groups and how to use them.
- [Chapter 4, "Managing paths"](#)  
A companion chapter to [Chapter 2, "Creating and managing projects,"](#) this chapter describes how paths are used in JBuilder. Describes how to work with libraries and packages.
- [Chapter 5, "Compiling Java programs"](#)  
Explains how to compile your project and set compiler options. Also explains how to compile from the command line.
- [Chapter 6, "Building Java programs"](#)  
Explains the JBuilder build process. Discusses the difference between Make and Rebuild. Describes how to build external Ant files, how to build project groups, and how to use additional JBuilder features, such as automatic source packages, package filtering, and resource copying.

- [Chapter 7, “Running Java programs”](#)  
Explains how to use JBuilder’s IDE to run your applications and applets. Also explains how to manage runtime configurations.
- [Chapter 8, “Debugging Java programs”](#)  
Explains how to use JBuilder’s integrated debugger to find and fix errors in your program. Describes the entire debugging process, discusses the types of bugs you may encounter, and explains how to examine the values of program variables to uncover bugs.
- [Chapter 9, “Remote debugging”](#)  
Describes how to debug a program running on a remote computer.
- [Chapter 10, “Creating JavaBeans with BeansExpress”](#)  
Describes how to create a JavaBean and how to convert an existing class to a JavaBean.
- [Chapter 11, “Visualizing code with UML”](#)  
Describes how to use JBuilder’s code visualization features to examine, navigate, and understand your code.
- [Chapter 12, “Refactoring code symbols”](#)  
Describes how to use JBuilder’s refactoring features.
- [Chapter 13, “Unit testing”](#)  
Describes the unit testing features available in JBuilder.
- [Chapter 14, “Creating Javadoc from API source files”](#)  
Describes how to use JBuilder’s Javadoc-related features to generate HTML formatted output files from comments in API source code.
- [Chapter 15, “Deploying Java programs”](#)  
Provides an overview of general deployment issues and explains how to create Java archive files with the `jar` tool.
  - [“Deploying with the Archive Builder” on page 15-17](#)  
Explains how to use the Archive Builder to deploy your Java programs.
  - [“Creating executables with the Native Executable Builder” on page 15-29](#)  
Explains how to use the Native Executable Builder to create native executables for your deployed Java programs.



- [Chapter 16, “Internationalizing programs with JBuilder”](#)  
Explains how to internationalize your Java application or applet with JBuilder.
- Tutorials:
  - [Chapter 17, “Tutorial: Compiling, running, and debugging”](#)  
Find and fix syntax errors, compiler errors, and logic errors.
  - [Chapter 18, “Tutorial: Building with Ant files”](#)  
Use an Ant build file to build a project.
  - [Chapter 19, “Tutorial: Remote debugging”](#)  
Use remote debugging features to attach to a program already running on a remote computer and debug using cross-process stepping.
  - [Chapter 20, “Tutorial: Visualizing code with the UML browser”](#)  
Use JBuilder’s UML features to visualize, analyze, and troubleshoot your code.
  - [Chapter 21, “Tutorial: Creating and running test cases and test suites”](#)  
Use JBuilder’s unit testing features to create and run unit tests with JUnit.
  - [Chapter 22, “Tutorial: Working with test fixtures”](#)  
Create a JDBC Fixture and a Comparison Fixture and use them in a test case.

The following are appendixes to *Building Applications with JBuilder*:

- [Appendix A, “Creating configuration files for native executables”](#)  
Learn how to write custom configuration files to launch the native executables you create with the Native Executable Builder or the Archive Builder.
- [Appendix B, “Using the command-line tools”](#)  
Explains how to use JBuilder’s command-line compilers, JBuilder’s command-line arguments, and the JDK tools. Also discusses setting the class path.

You can also find the following topics in online help:

- “Error and warning messages”

Describes the kinds of errors and warnings that can occur when compiling, debugging, or running JBuilder applications. Also includes a list of errors by number.

- “Compiler error messages”

Lists the error messages by number and includes a description of each message.

For definitions of any unfamiliar Java terms, see “Online glossaries” in *Getting Started with Java*.

## Documentation conventions

---

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

**Table 1.1**    Typeface and symbol conventions

Typeface	Meaning
Monospaced type	<p>Monospaced type represents the following:</p> <ul style="list-style-type: none"> <li>• text as it appears onscreen</li> <li>• anything you must type, such as “Type <code>Hello World</code> in the Title field of the Application wizard.”</li> <li>• file names</li> <li>• path names</li> <li>• directory and folder names</li> <li>• commands, such as <code>SET PATH</code></li> <li>• Java code</li> <li>• Java data types, such as <code>boolean</code>, <code>int</code>, and <code>long</code>.</li> <li>• Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events</li> <li>• argument names</li> <li>• field names</li> <li>• Java keywords, such as <code>void</code> and <code>static</code></li> </ul>
<b>Bold</b>	<p>Bold is used for java tools, <code>bmj</code> (Borland Make for Java), <code>bcj</code> (Borland Compiler for Java), and compiler options. For example: <b><code>javac</code></b>, <b><code>bmj</code></b>, <b><code>-classpath</code></b>.</p>
<i>Italics</i>	<p>Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.</p>
<i>Keycaps</i>	<p>This typeface indicates a key on your keyboard, such as “Press <i>Esc</i> to exit a menu.”</p>
[ ]	<p>Square brackets in text or syntax listings enclose optional items. Do not type the brackets.</p>

**Table 1.1** Typeface and symbol conventions (continued)

Typeface	Meaning
< >	<p>Angle brackets are used to indicate variables in directory paths, command options, and code samples.</p> <p>For example, &lt;filename&gt; may be used to indicate where you need to supply a file name (including file extension), and &lt;username&gt; typically indicates that you must provide your user name.</p> <p>When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (&lt; &gt;). For example, you would replace &lt;filename&gt; with the name of a file, such as <code>employee.jds</code>, and omit the angle brackets.</p> <p><b>Note:</b> Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as &lt;font color=red&gt; and &lt;ejb-jar&gt;. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.</p>
<i>Italics, serif</i>	<p>This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, &lt;url="jdbc:borland:jbuilder\\samples\\guestbook.jds"&gt;</p>
...	<p>In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.</p>

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

**Table 1.2** Platform conventions

Item	Meaning
Paths	<p>Directory paths in the documentation are indicated with a forward slash (/).</p> <p>For Windows platforms, use a backslash (\).</p>
Home directory	<p>The location of the standard home directory varies by platform and is indicated with a variable, &lt;home&gt;.</p> <ul style="list-style-type: none"> <li>For UNIX and Linux, the home directory can vary. For example, it could be <code>/user/&lt;username&gt;</code> or <code>/home/&lt;username&gt;</code></li> <li>For Windows NT, the home directory is <code>C:\Winnt\Profiles\&lt;username&gt;</code></li> <li>For Windows 2000 and XP, the home directory is <code>C:\Documents and Settings\&lt;username&gt;</code></li> </ul>
Screen shots	<p>Screen shots reflect the Metal Look &amp; Feel on various platforms.</p>

## Developer support and resources

---

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

### Contacting Borland Technical Support

---

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support on installation of the Borland product to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at <http://www.borland.com/devsupport/>, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

### Online resources

---

You can get information from any of these online sources:

**World Wide Web**     <http://www.borland.com/>

**FTP**     <ftp://ftp.borland.com/>

Technical documents available by anonymous ftp.

**Listserv**     To subscribe to electronic newsletters, use the online form at:

<http://info.borland.com/contact/listserv.html>

or, for Borland's international listserver,

<http://info.borland.com/contact/intlist.html>

### World Wide Web

---

Check [www.borland.com/jbuilder](http://www.borland.com/jbuilder) regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- <http://www.borland.com/jbuilder/> (updated software and other files)
- <http://www.borland.com/techpubs/jbuilder/> (updated documentation and other files)
- <http://community.borland.com/> (contains our web-based news magazine for developers)

## Borland newsgroups

---

You can register JBuilder and participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at <http://www.borland.com/newsgroups/>.

## Usenet newsgroups

---

The following Usenet groups are devoted to Java and related programming issues:

- `news:comp.lang.java.advocacy`
- `news:comp.lang.java.announce`
- `news:comp.lang.java.beans`
- `news:comp.lang.java.databases`
- `news:comp.lang.java.gui`
- `news:comp.lang.java.help`
- `news:comp.lang.java.machine`
- `news:comp.lang.java.programmer`
- `news:comp.lang.java.security`
- `news:comp.lang.java.softwaretools`

**Note** These newsgroups are maintained by users and are not official Borland sites.

## Reporting bugs

---

If you find what you think may be a bug in the software, please report it in the [Support Programs](http://www.borland.com/devsupport/namerica/) page at <http://www.borland.com/devsupport/namerica/>. Click the “Reporting Defects” link to bring up the Entry Form.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may e-mail [jpppubs@borland.com](mailto:jpppubs@borland.com). This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

## Creating and managing projects

JBuilder does everything within the context of a *project*. As used in this documentation, the term “project” includes all the files within a user-defined body of work, the directory structure those files reside in, and the paths, settings, and resources required.

The project is an organizational tool, not a repository. This means that files in a project can be in any folder. Restructuring a project tree has no effect on your directory tree. This gives you independent control of projects and directory structure.

Each project is administered by a project file. The project file’s name is the name of the project with a `.jpx` extension. The project file contains a list of files in the project and maintains the project properties, which include a project template, default paths, class libraries, and connection configurations. JBuilder uses this information when you load, save, build, or run a project. Project files are modified whenever you use the JBuilder development environment to add or remove files or set or change project properties. You can see the project file as a node in the project pane. Listed below it are all the packages and files in the project.

**Note**  
This is a feature of  
JBuilder SE and  
Enterprise

If automatic source packaging is enabled, source package nodes also appear in the project pane. These display files and packages that are on the project’s source path. See [“Automatic source packages” on page 6-21](#).

While you can include any type of file in a JBuilder project, there are certain types of files that JBuilder automatically recognizes and for which it has appropriate views. You can add binary file types, customize file type handling, and see the icons associated with file types by selecting Tools | IDE Options and choosing the File Types tab.

You’re asked to configure file associations when starting JBuilder for the first time. JBuilder prompts you to associate `.class`, `.java`, and project and project group file types of files with JBuilder. Doing so makes JBuilder the

default program for opening and viewing these files. You can change these configurations by selecting Tools | Configure File Associations to invoke the Configure File Associations dialog box.

## Creating a new project

---

To start a new project, use JBuilder's Project wizard to generate the basic framework of files, directories, paths, and preferences automatically. The Project wizard can automatically create a project notes file for your notes and comments. The class Javadoc fields that are filled out in the Project wizard are used in the project notes file, as Javadoc header comments when using JBuilder's wizards to create Java files, and consequently included in Javadoc-generated documentation. These comments can be modified on the General page of the Project Properties.

When using many of JBuilder's wizards, if a project is not open, the Project wizard is launched first so that you can create a new project.

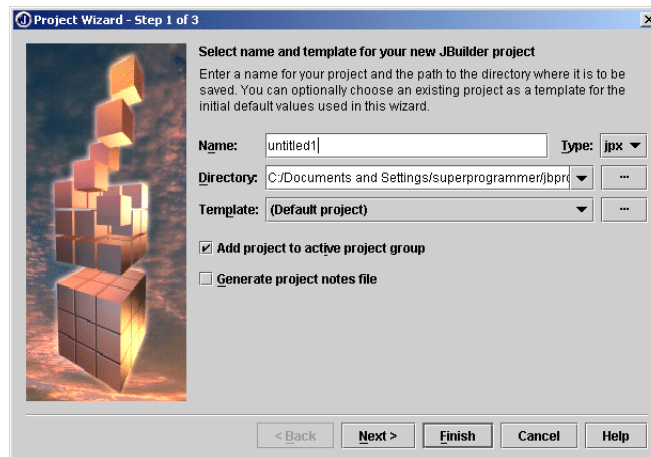
## Creating a new project with the Project wizard

---

To create a new project with the Project wizard, select File | New Project. You may also choose File | New, select the Project tab, and double-click the Project icon. The Project wizard appears.

### Selecting project name and template

Use Step 1 to set your project name, type, root directory, and project template.





**1** Enter a name for the new project.

JBuilder uses the project name as the package name by default.

Any file name legal to the file system is allowed for the project name. However, there are other names which are derived from this file name and these derived names have restrictions which must be met:

- a** The project directory name can appear on a Java classpath. Since embedded spaces can cause problems, if there are spaces, they are replaced with underbars.
- b** The wizard uses the project name as the default package name. Therefore, it must be a legal Java package name. This means that leading numbers are removed from the file name, spaces are replaced with underbars, the case is forced to lowercase, and the name may be truncated if it's too long.

**2** Select the project directory. The project directory is the one that contains the project file. Many other project paths, such as the source and backup paths, descend from this by default.

- Click the down arrow to select a directory you've used previously as the parent or to choose one in the same tree that you can edit.
- You can edit the field directly or click the ... button to browse to an existing directory.

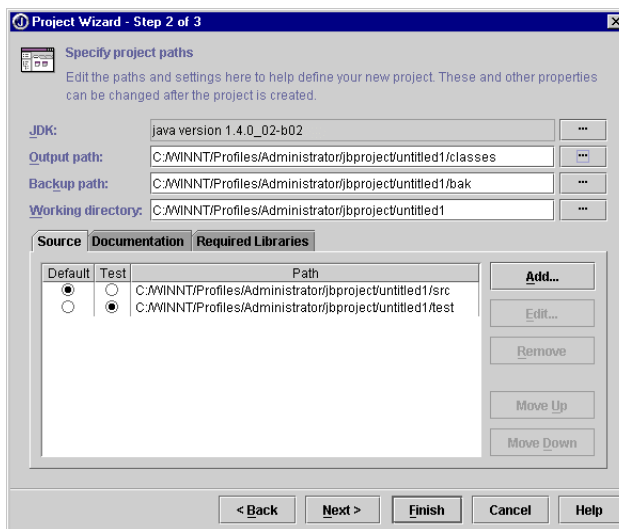
**Note** If you enter a path that's syntactically flawed, you won't be able proceed.

**3** Accept (Default project) as the value of the Template field. (You can click the Help button to read about project templates if you like.)**4** To add the project you are creating to an existing project group (the project group must be active), check the Add Project To Active Project Group check box. This check box is enabled only if you currently have an open and active project group. Project groups are available in JBuilder Enterprise only. For more information about project groups, see [Chapter 3, "Working with project groups."](#)**5** To generate an HTML project notes file, check the Generate Project Notes File check box. This file is optional.**6** Click Next to go to Step 2.

If the Finish button is enabled, you can click it, accepting JBuilder's defaults for the rest of the wizard, and create the project immediately.

## Setting project paths

Step 2 sets all paths for the project, including the JDK version to compile against. You can change these settings later using the Paths page of the Project Properties dialog box (Project | Project Properties) if you need to.



JBuilder suggests the project directory set in Step 1 as the working directory. The working directory is the starting directory that JBuilder gives a program when it is launched. Any directory may be configured as the working directory.

To change any of paths on this page, either type in the new path or navigate to it by clicking the ... button next to the appropriate field.

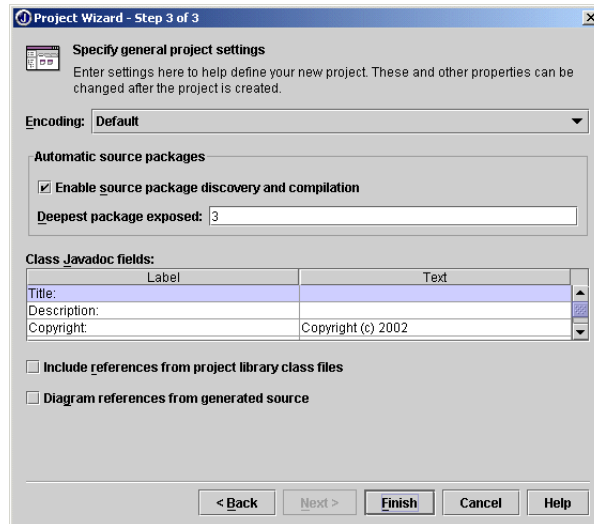
**Note** If you're just beginning with JBuilder, simply accept the default values on this page. If you enter a path that's syntactically flawed, you can't proceed. More advanced users might want to change these directories to those of their choosing. For more information about using this page and complete information about the various directories, click the Help button in the Project wizard.

Click Next to go to Step 3 or click Finish to create your project. More advanced users might want to continue to Step 3.

## Setting general project settings

Step 3 of the Project wizard includes general project settings, such as encoding, default runtime configuration, automatic source packages, class Javadoc fields, and references from project libraries.

This information can later be changed on the General page and Run page of the Project Properties dialog box (Project | Project Properties).



- 1 Choose an encoding or accept the default encoding. Encoding determines how JBuilder should handle characters beyond the ASCII character set. The default is your platform's default encoder.

### See also

- [“Specifying a native encoding for the compiler” on page 16-12](#)
- [“Internationalization Tools: native2ascii” at http://java.sun.com/products/jdk/1.4/docs/tooldocs/tools.html#intl](http://java.sun.com/products/jdk/1.4/docs/tooldocs/tools.html#intl)

- 2 Choose the Enable Assert Keyword option if you want `assert` recognized as a keyword.

JBuilder supports JDK 1.4 assertions. Before JDK 1.4, `assert` was a keyword reserved for future use. With JDK 1.4, the `assert` keyword has been added to the language and is used in the assertion facility. `assert` takes a boolean value which checks a condition before the associated expression is executed. Assertions are enabled or disabled at runtime.

### See also

- <http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>

This is a feature of  
JBuilder SE and  
Enterprise

**3** Select Automatic Source Packages options.

- a** The Enable Source Package Discovery And Compilation option is enabled by default. When it's enabled, several things happen:
- All packages in the project's source path appear in the project pane (upper left pane) of the IDE.
  - Packages that contain Java files are compiled automatically.
  - Files generated by this compilation are copied to the project's out path.

Not all packages are displayed, only a logical subset determined by how deeply you tell JBuilder to expose packages.

**b** Select how deeply you want packages exposed.

JBuilder exposes packages to the level you set, unless adding more levels to a package will not change the length of the package list. For instance, if you have these three packages,

```
one.two.three.four
one.two.three.five
one.two.four.six
```

and set the package level to three, this is what shows in the project pane:

```
one.two.three
one.two.four.six
```

The two packages `one.two.three.four` and `one.two.three.five` are both contained in `one.two.three`, so JBuilder represents only the parent package and allows you to expand the package node to access the packages and files inside.

The package `one.two.four.six` is exposed to the fourth level because shortening the representation won't shorten the package list, so you might as well see what you've got.

**See also**

- [“Packages” on page 4-6](#)

**4** Specify the class Javadoc fields. These can be used in the project notes file, your application's Help | About dialog box, and can also be inserted as Javadoc header comments in wizard-generated files created for your project.

Select a field to edit and enter the appropriate text in the Text column.

Find References is a feature of JBuilder SE and Enterprise

UML is a feature of JBuilder Enterprise

- 5 Check Include References From Project Library Class Files if you want to be able to find references to any of the project libraries. The Find References command on the editor's context menu allows you to discover all source files that use a given symbol. Also check this if you want your project's UML diagrams to show references from project libraries.

#### See also

- [“Finding references to a symbol” on page 12-8](#)
- [Chapter 11, “Visualizing code with UML”](#)

- 6 If you have JBuilder Enterprise and you want to include references such as IIOP files or EJB stubs in the UML diagrams of your project, check the Diagram References From Generated Source option.

#### See also

- [Chapter 11, “Visualizing code with UML”](#)

- 7 Click Finish when done.

JBuilder creates a new project that appears in the project pane.

If you later want to change the settings you specified for your project using the Project wizard, you can change them using the Paths page and the General page of the Project Properties dialog box (Project | Project Properties).

## Creating a project from existing files

---

This is a feature of JBuilder SE and Enterprise

The Project For Existing Code wizard allows you to create a new JBuilder project using an existing body of work. When you use this wizard, JBuilder scans the existing directory and builds paths that are used for compiling, searching, debugging, and other processes. Any JAR or ZIP files that aren't already in libraries are placed in a new library and added to the project. Project libraries are listed on the Required Libraries tab of the Paths page of Project Properties (Project | Project Properties).

To access the Project For Existing Code wizard,

- 1 Select File | New. The object gallery appears.
- 2 Select the Project tab.
- 3 Double-click the Project For Existing Code icon.

## Selecting the source directory and name for your new JBuilder project

---

Step 1 sets your project directory, name, type, and project template.

- 1 Choose the directory where the existing project or source tree is located. Click the ... button to browse to it. JBuilder scans the selected directory for such files as class, source, JAR, and ZIP and places them in the appropriate directories within that directory.
- 2 Enter a name for the new project. JBuilder uses the project name as the package name by default, so if you have an existing package name use that as the project name. JBuilder's wizards also use the project name as the package name, which can be edited in the wizards.
- 3 Select your project template:
  - Click the down arrow to choose a project you've used previously as a template.
  - Click the ... button to use a different project as the template in the Open Project dialog box.

The project template provides default values for the settings described in the Project Properties dialog box (Project | Project Properties). If you already have a JBuilder project whose project properties are close to what is required in the new project, select it here. This minimizes the repetitive work involved in setting up a new project within an established environment.

- 4 Choose whether to generate an HTML project notes file. The initial information in this file, such as title, author, and description, is generated from the class Javadoc fields set in Step 3 of the Project For Existing Code wizard. You can also add notes and other information in this file as needed.
- 5 Click Next to go to Step 2.

Steps 2 and 3 of the Project For Existing Code wizard are identical to the Project wizard. These steps are also the same as the Paths page and the General page of Project Properties. See [“Creating a new project with the Project wizard” on page 2-2](#).

If your project requires specific libraries, you can add them to the project on the Paths page of the Project Properties dialog box. To set the main class to run your project, choose the Run page of the Project Properties dialog box.

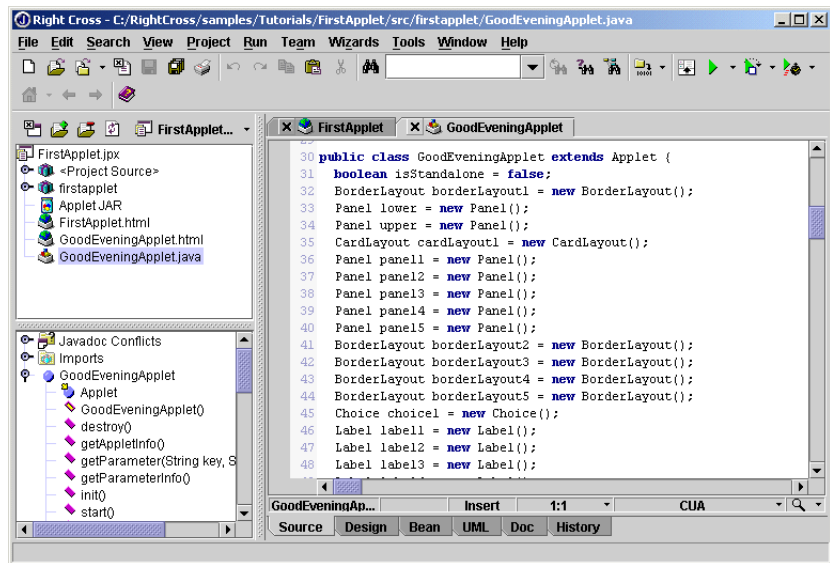
### See also

- [“Setting paths for required libraries” on page 2-22](#)
- [“Using the Run command” on page 7-3](#)

## Displaying files

JBuilder displays each open file of a project in the content pane of the AppBrowser. Double-click a file in the project pane to open it in the content pane. A tab with the file name appears at the top of the content pane.

The following figure shows a project file, `Welcome.jpx`, in the project pane with the source files listed below it. This project contains a package and three source files. Two files are open in the content pane with the selected file, `WelcomeApp.java`, showing in the source pane.



Right-click the project file to display a menu with such menu selections as Open, Add Files/Packages, Remove From Project, Close Project, Make, Rebuild, and Properties. Many of these menu selections are also available from the Project menu.

## Switching between files

When you have a lot of files open, it's not always easy to look through all the open file tabs to find the one you want to use. There are two ways to switch quickly between open files:

- Choose Window | Switch or press **Ctrl+B** to display the Switch dialog box, which lists all the open files in your project. Scroll down to select the file you want. Or begin typing the name of the file and the first match in the list is selected; keep typing until the file you want is selected. Once you've selected your file, choose OK.

- Choose Window to display the Window menu and select the file you want from the list of open files listed on the menu. The currently active file has a checkmark next to it.

You can also switch between open *projects* from the Window menu.

**Tip** If you prefer, you can have the file tabs display vertically on the right side of the content pane instead of at the top. Choose Tools | IDE Options and in the Content Pane Tab options, select Vertical from the drop-down Orientation list. The Content Pane Tab options also offer other tab display options you might like to explore.

## Saving projects

---

When you are working on a project, you can save it to the suggested location or to a directory of your choice. By default, JBuilder saves projects to the `jbproject` directory of your home directory, although this depends on how your system is set up. Each project is saved to its own directory within `jbproject`. Each project directory includes a project file, an optional `.html` file for project notes, a `classes` subdirectory for generated files (such as `.class` files), a `src` subdirectory for source files, a `bak` subdirectory for backup files, and a `doc` directory for documentation.

### Saving and closing projects



To save a project, select File | Save All, File | Save Project, or click the Save All button on the main toolbar.



To close a project, select File | Close Projects or click the Close Project button on the project toolbar.

### See also

- [“How JBuilder constructs paths” on page 4-9](#)
- [“Where are my files?” on page 4-12](#)

## Opening an existing project

---

There is one way to open an existing project for the first time: use File | Open Project. There are two ways to open an existing project you have opened before: either the File | Open Project command or the File | Reopen command.

To open a project using the File | Open Project command,

- 1 Choose File | Open Project. The Open File dialog box appears.



- 2 Navigate to the directory that contains the project file you want to open.
- 3 Select the project file and click OK or press *Enter*. You can also double-click the project file to open it.

To open a previously opened project with the File | Reopen command,

- 1 Choose File | Reopen.
- 2 Choose the project you want to open from the list of previously opened projects.

The project file and its source files will appear in the project pane.

To open a file in the content pane, you may do one of three things:

- Double-click the file in the project pane.
- Select the file in the project pane and press *Enter*.
- Right-click the file in the project pane and select Open.

To view a project in a new AppBrowser,

- 1 Select Window | New Browser.
- 2 Select File | Open Project and navigate to the file in the Open Project dialog box.

If you have more than one AppBrowser open and the same files open in multiple AppBrowsers, changes you make in one AppBrowser will be reflected immediately in the same file open in the other AppBrowser. This keeps all your working versions of a file congruent.

**Note** All open projects are available in all AppBrowsers from the Project drop-down list.

## Creating a new Java source file

---

There are several ways to create Java source files within JBuilder. Many of JBuilder's wizards create files. Most of these are available from the object gallery (File | New) or from the Wizards menu. Specifically, the Class wizard generates the framework of a new Java class.

To create an empty Java source file,

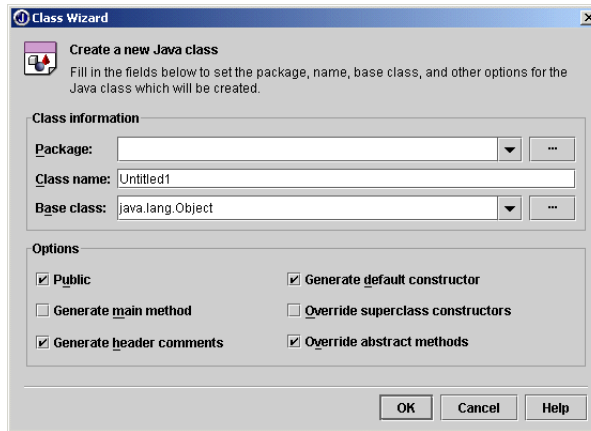
- 1 Choose File | New File to display the Create New File dialog box.
- 2 Type the name of the file in the Name field.
- 3 Select the `java` file type from the drop-down list or include the extension when you type the name.

- 4 If you want to change the directory where the file is saved, type in the new directory in the Directory field or choose the ... button to select the directory you want.
- 5 Choose OK.

JBuilder creates the new file and opens it in the content pane.

To create a new Java source file using the Class wizard,

- 1 Create a new project as described in [“Creating a new project” on page 2-2](#).
- 2 Choose File | New Class to display the Class wizard.



- 3 Enter the package, class, and base class names in the Class wizard.
- 4 Select options for exposure, method handling, and header comments.
- 5 Click OK.

The .java file is created and added to your project (its node appears in the project pane). The new file opens in the content pane in editor.

### See also

- [“Adding to a project” on page 2-13](#)
- [“Packages” on page 4-6](#)
- [“Setting general project settings” on page 2-4](#)



After you edit a file, save it by choosing File | Save or clicking the Save File icon. The path and parent directory of the file appears at the top of the AppBrowser window when the file is selected and open. You can save all files in your project by choosing File | Save All or clicking the Save All icon.



# Managing projects

---

JBuilder is designed to support the developer in performing as many development tasks as possible. JBuilder's project tools, rich IDE, and extensive editor features automate and simplify the work of development while allowing you to focus on your code.

## Adding to a project

---

In the AppBrowser, you can add folders, new files, and existing files to your project. Many of these commands are available from the project pane's context-sensitive (right-click) menu as well as the main menu.

For larger projects, you can use project folders to organize your project's hierarchy.

**Note** Project folders are for organizational purposes only and do not correspond to directories on disk.

## Adding folders

Project folders don't affect the directory tree. They are organizing tools that allow you to sort elements of a project in a way that's useful to you without affecting the directory structure.

To add a project folder to a project,

- 1 Select the project in the project pane (upper left).
- 2 Choose File | New Folder from the main menu or right-click in the project pane.
- 3 Select New Folder.

**Tip** To nest the new folder inside an existing one, select the existing folder before choosing File | New Folder.

- 4 Type the name of the folder.
- 5 Click OK or press *Enter*.

To add a file to a folder,

- 1 Either right-click the folder and choose Add Files/Packages to open the Add Files Or Packages To Project dialog box, or select the folder and click the Add Files/Packages button on the project pane toolbar.
- 2 Navigate on the Explorer page of the Add Files To Project dialog box to the directory that contains the file you want to add.
- 3 Select the file and click Open.



## Adding files and packages

Automatic source path discovery, a feature of JBuilder SE and Enterprise, adds new files and packages to the project on the fly. Set this option on the General page of the Project Properties dialog. It's on by default.

If you're not using this feature, files and packages must be explicitly added to a project in order for JBuilder to treat them as part of the project. Add files and packages to the current project using the Add Files Or Packages To Project dialog box.

There are two ways to access this dialog box. Use the way you prefer:



- Click the Add Files/Packages button that's on the project pane toolbar.
- Right-click any node in the project pane and choose Add Files/Packages from the context menu.

When the dialog box is appears, follow these steps:

- 1 Select the Explorer page to add a file, the Packages page to add a package, or the Classes page to add a class. All of these pages support multiple selection.
- 2 Navigate to the file, class, or package you want to import.
- 3 Select the file or package you want. Once you're in a file's parent directory, you may type in the filename instead of selecting it.
- 4 Double-click your selection, click OK, or press *Enter*.

The new node appears inside the project directory in the project pane.

## Removing material from a project

---

You can remove folders, files, classes, and packages from your project without deleting them from the drive with the Remove From Project dialog box.

To remove a node from your project, choose one of these options:

- Right-click the node you want to remove, choose Remove From Project, and click OK.
- Select the node you want to remove, click the Remove From Project button on the project pane toolbar, and click OK.



**Note** If a folder contains files, they are also removed from the project.

You can also select multiple nodes and remove them from the project.

## Deleting material

---

You can also delete unwanted files, classes, and packages from the disk by right-clicking the item in the project pane and choosing Delete.

**Caution** This permanently deletes the files from the project *and* the computer's hard disk.

## Opening a file outside of a project

---

Use the File | Open file command to open a file in the AppBrowser without adding the file to the open project. A project must be open in order for this option to be available.

To open a file without adding it to a project,

- 1 Choose File | Open File. The Open File dialog box appears.
- 2 Select the file you want to open.
- 3 Click OK. The file contents are displayed in the AppBrowser.

This is a feature of  
JBuilder SE and  
Enterprise.

You can also open a file in the open project in another AppBrowser.

- 1 Right-click a file in the project pane.
- 2 Choose Open In New Browser.

## Renaming projects and files

---

To rename a project or file,

- 1 Select the project or file in the project pane.
- 2 Select Project | Rename, File | Rename, or right-click in the project pane and select Rename.
- 3 Enter the new name in the Rename dialog box and click OK.

You can also rename an open file using the file's tab at the top of the content pane:

- 1 Right-click the file tab at the top of the content pane.
- 2 Select Rename.
- 3 Enter the new name in the Rename dialog box and click OK.

**Note** Renaming a file does not change the file type. To change the file extension, use File | Save As.

**Caution** Renaming projects and files does not change the package and file names referenced inside the code. JBuilder SE and Enterprise provide rename refactoring. This changes all of the uses of the old name to match the new name.

**See also**

- [Chapter 12, “Refactoring code symbols”](#)

## Adding a new directory view

---

This is a feature of  
JBuilder SE and  
Enterprise

You can choose to add a new node to the project pane that can point to any directory of your choosing. For example, you might have code that is buried deep in your project structure. You can simply add the directory that contains that code as a project node in the project pane. When you open that node, you'll have immediate access to your code instead of having to navigate through a complicated directory structure.

A directory view is much like a live view of your directory or directory tree that displays all file types. Once you've created a directory view and when changes occur to the actual contents of the directory or its subdirectories, the directory view will be updated in the project pane (you might need to click the Refresh icon on the project toolbar to refresh the project).

When you pull a project using CVS, a new directory view node is created that shows the entire project directory tree with CVS subdirectories hidden. This provides the ability to locate any file in the project regardless of type, and select it for doing a CVS operation.

To add a directory view to a project,

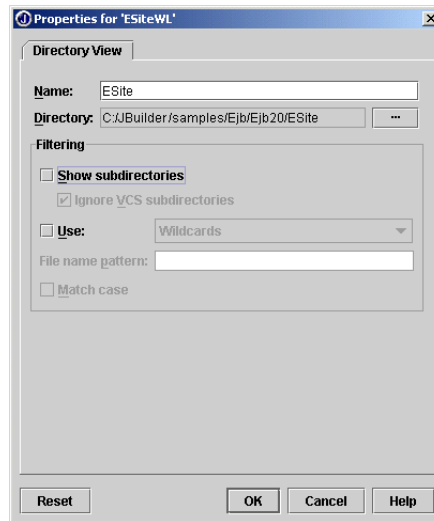
- 1 Choose Project | New Directory View or right-click the project node (the .jpx file) in the project pane and choose New Directory View.
- 2 Navigate to the directory you want to expose directly in the project pane.
- 3 Click OK and the directory you selected is added as a node in the project pane.
- 4 Open the node to see the files contained in the directory.

You can customize which files and subdirectories appear when you open the directory view project node. For example, you could choose to display just .java files or .html files. You can even have multiple directory views of the same project so that one view displays .java files, another .html files, and another displays all files that begin with the letter 'a'.

To customize the directory view,

- 1 Right-click the directory view project node in the project pane.
- 2 Choose Properties.

The Properties dialog box appears containing just a Directory View page:



- 3 Specify a meaningful name for the directory view project node or keep the default name, which is the name of the directory.
- 4 Use the Filtering options to filter the files and directories that appear when you open the directory view project node. For example, These are examples of file patterns:

```
*.java
myfile?.java
file??.*
```

**Note** You cannot string multiple file patterns together, such as `*.java;*.cpp`.

For complete information about using these options, click the Help button in this dialog box.

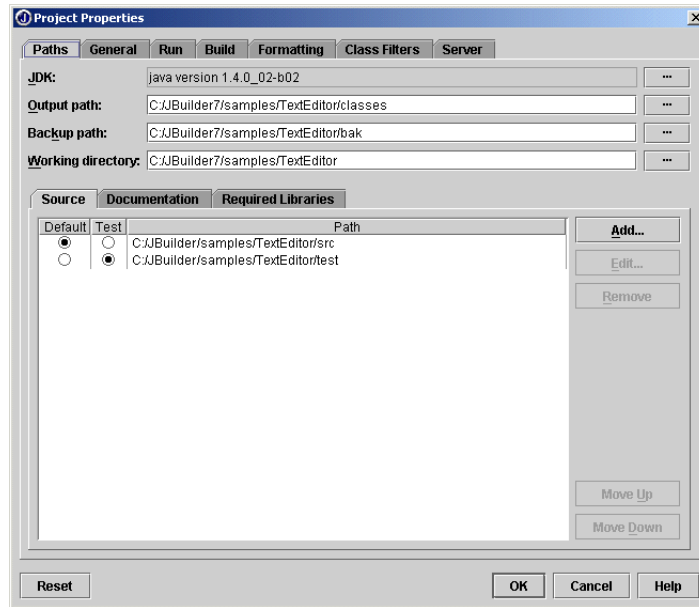
- 5 Click OK.

## Setting project properties

Project properties control how the project is compiled. Using the Project Properties dialog box (Project | Project Properties), you can specify project path settings, set up a run configuration for your project, specify how a project is built, customize the display of UML diagrams, specify server options, and much more.

To set project properties,

- 1 Right-click the `.jpx` file name in the project pane and choose Properties to display the Project Properties dialog box. Or, select the project and choose Project | Project Properties. The Project Properties dialog box appears.
- 2 Select the appropriate tab for the options you want to set. In this image the Paths page is selected:



Here are brief descriptions of what you can do on each page of the Project Properties dialog box:

- Paths page: set project path settings for the JDK version, output path, backup path, working directory, source paths, test path, documentation path, and required libraries paths.
- General page: set options for encoding, enabling automatic source packages, modify class Javadoc fields that wizards can generate, and set the option to include references from project libraries.
- Run page: select or create a configuration to use for running or debugging.
- Build page: set compiler options for building a project, including debug information and selective resource copying.
- Formatting page: set code formatting options. JBuilder can speed your coding by formatting it automatically to your specifications. This is a feature of JBuilder Enterprise.



- Class Filters: set options for class filters.
- Server page: set server options. This is a feature of JBuilder Enterprise.

**Note**

You can also set these options for all future projects in the Default Project Properties dialog box (Project | Default Project Properties) or select the default project as your project template in the Project wizard.

## Setting the JDK

---

On the Paths page, you can set the JDK version, various paths for the project, and the Required Libraries Paths.

JBuilder SE and Enterprise fully support JDK switching, while JBuilder Personal allows you to edit a single JDK. JBuilder compiles and runs against all Sun JDKs and many others.

With JBuilder SE and Enterprise, you can set the JDK version for your project on the Paths page of the Project Properties dialog box as well as add, edit, and remove JDKs in the Configure JDKs dialog box. See [“Setting the JDK in SE and Enterprise” on page 2-20](#).

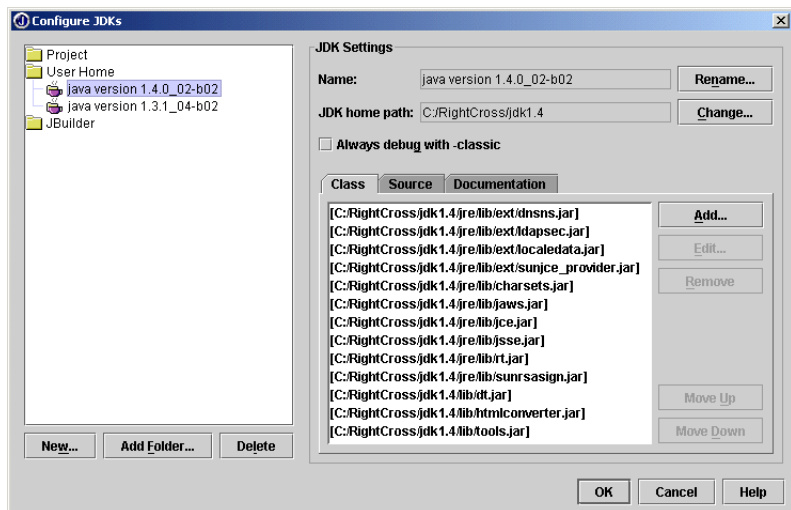
For JBuilder Personal, you can edit the JDK in the Configure JDKs dialog box (Tools | Configure JDKs). See [“Editing the JDK” on page 2-19](#).

## Editing the JDK

---

You can edit the current JDK version as follows:

- 1 Select Tools | Configure JDKs to open the Configure JDKs dialog box:



- 2 Click the Change button to the right of the JDK Home Path field. The Select Directory dialog box appears.

- 3 Browse to the target JDK.
- 4 Click OK to change the JDK.

Note the revised JDK name and home path in the Configure JDKs dialog box.

- 5 Click OK to close the Configure JDKs dialog box.

## Debugging with -classic

The Always Debug With -Classic option in the Configure JDKs dialog box provides improved performance for users with JVM versions below 1.3.1. JBuilder automatically checks to see if this option will improve your performance, then checks or unchecks this box according to what will give you the best results. This feature is available in all editions of JBuilder.

In performing its evaluation, JBuilder performs two checks:

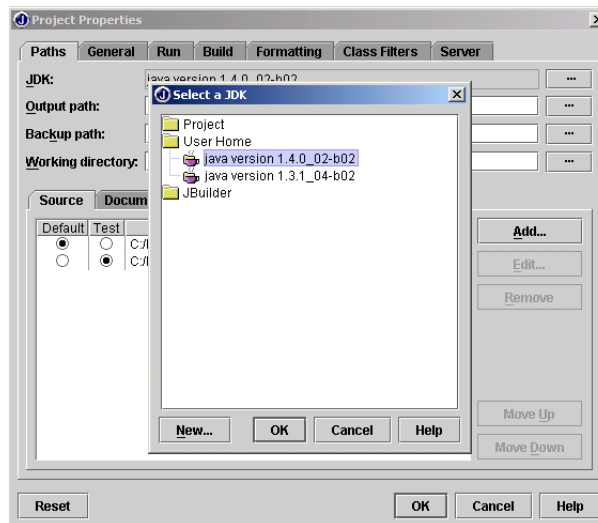
- 1 Do you have the Classic VM?
- 2 If present, is the JVM a version earlier than 1.3.1?

This selection is overridden when you define VM parameters such as `native`, `hotspot`, `green`, or `server`.

## Setting the JDK in SE and Enterprise

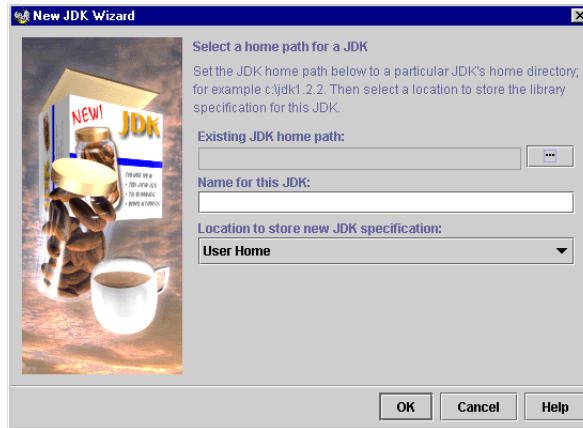
JBuilder SE and Enterprise support JDK switching. You can also add, edit, and delete JDKs. To switch to another JDK, follow these steps:

- 1 Select Project | Project Properties and select the Paths tab.
- 2 Click the ... button to the right of the JDK version. The Select A JDK dialog box appears:



### 3 If the target JDK is listed, select it and press OK.

If it's not listed, select New to open the New JDK wizard.



- a Click the ... button and browse to the home directory of the JDK you want to add to the list. Click OK. Note that the JDK Name field is filled in automatically.
- b Select the location to store the JDK specifications:
  - **User Home:** saves the JDK specifications in a `.library` file in the `.jbuilder` directory of the user's home directory. Save to this location if you want the JDK available to all projects.
  - **JBuilder:** saves the JDK specifications in a `.library` file in the `jbuilder` directory. Multiple users who are using JBuilder on a network or sharing JBuilder on a single machine have access to the JDKs in this folder. This is a feature of JBuilder SE and Enterprise.
  - **Project:** saves the JDK specifications in a `.library` file in the current project's directory. Save to this location if you only want the JDK available to this project. This is a feature of JBuilder SE and Enterprise.
  - **User-defined folder:** saves the JDK specifications to an existing user-defined folder or shared directory. You must add the new folder (select Tools | Configure JDK and click Add Folder) before it can appear in the drop-down list. This is a feature of JBuilder Enterprise.
- c Click OK. Note that the JDK specification has been added to the specified directory in the Select A JDK dialog box.
- 4 Click OK to close the Select A JDK dialog box. Note that the JDK path is updated to the new selection.
- 5 Click OK to close the Project Properties dialog box.

**6** Save the project. The JDK version is updated in the project file.

**Tip** You can add, edit, and delete JDKs by selecting Tools | Configure JDKs. You can also modify the Default Project Properties (Project | Default Project Properties) to change the JDK for all future projects.

## Configuring JDKs

---

Adding and deleting  
JDKs are features of  
JBuilder SE and  
Enterprise

You can add, edit, and delete JDKs in the Configure JDKs dialog box (Tools | Configure JDKs). JBuilder Personal users can edit JDKs as explained in [“Editing the JDK” on page 2-19](#).

In this dialog box, you can

- Name the JDK by selecting the Rename button.
- Add, edit, remove, and reorder JDK class, source, and documentation files.
- Open the New JDK wizard and add JDKs by selecting the New button.
- Add a folder that others can share. This is a feature of JBuilder Enterprise.
- Delete an existing JDK from the list.

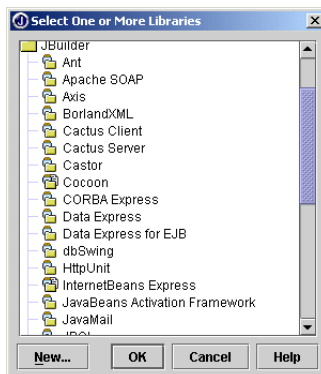
### See also

- Configure JDKs dialog box Help button.
- New JDK wizard Help button.

## Setting paths for required libraries

---

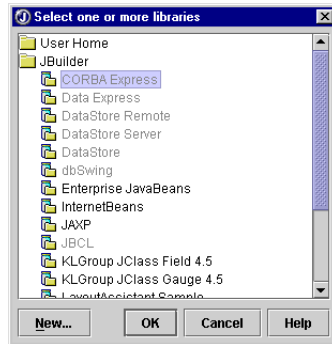
On the Paths page of the Project Properties dialog box, you can set the libraries to use when compiling. JBuilder places any selected libraries on the classpath. To add, edit, remove, and reorder libraries, select the Required Libraries tab.



You can select libraries in the Required Libraries list on the Paths page and edit, delete, or change their order in the library list.

**Note** Libraries are searched in the order listed. To switch the order of libraries, select a library, then click Move Up or Move Down.

The Add button displays the Select One Or More Libraries dialog box, where you choose the libraries to add to your project. Select New in this dialog box to open the New Library wizard and create a new library.



You can also configure libraries by selecting Tools | Configure Libraries.

### See also

- [“Working with libraries” on page 4-1](#)

## Working with multiple projects

---

You can work on multiple projects simultaneously in the JBuilder development environment. You can open them in one AppBrowser or in different AppBrowsers. All open projects are available from any open AppBrowser from the Project drop-down list. Any changes made in one AppBrowser are also made in any other open AppBrowsers displaying the same project. Open a new JBuilder AppBrowser by selecting Window | New Browser.

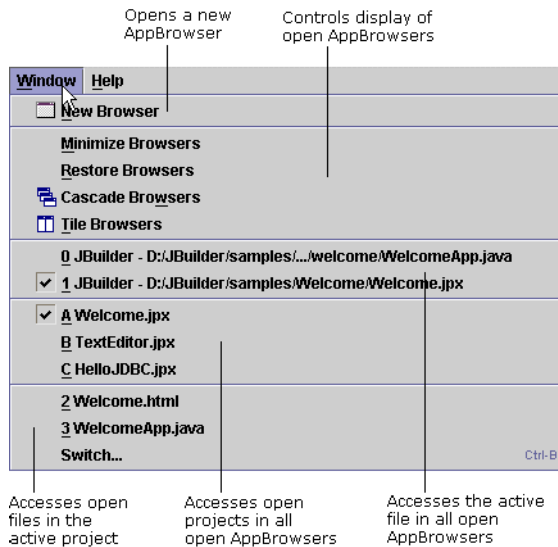
If you have JBuilder Enterprise, you can also group multiple projects in project groups. Project groups are particularly useful when you are working with related projects. For information about projects groups, see [Chapter 3, “Working with project groups.”](#)

### Switching between projects

---

If several projects are open in the AppBrowser, only one project is visible in the project pane. Switch to another open project by selecting the project from the Project drop-down list on the toolbar above the project pane.

Open AppBrowsers, projects, and files are all available from the Window menu:



Choose New Browser to open another AppBrowser. Control the browser windows with the next four commands. Select either the open file or the project file that belongs to an alternate AppBrowser to switch between browser windows.

Open files in the current browser are also available from this menu. This provides an alternate way to access open files—especially helpful for those who prefer not to show file names on file tabs.

## Saving multiple projects

To save changes to all open files and projects, choose File | Save All. All files in all open AppBrowsers are saved.

## More information about projects

While you are working with your JBuilder projects, you must have a good understanding of how JBuilder uses paths so you can make full use of features such as libraries and CodeInsight. See [Chapter 4, “Managing paths.”](#)

The ability to group projects is a feature of JBuilder Enterprise

JBuilder allows you to group projects into a project group. To read about project groups, see [Chapter 3, “Working with project groups.”](#)

## Working with project groups

This is a feature of  
JBuilder Enterprise

Project groups are containers for projects and can be useful when working with related projects. For example, you might have two projects that have dependencies on each other, such as a client and a server. Another logical grouping would be projects that use the same source files but have different settings, such as different target application servers or different JDKs. In addition, project groups provide other advantages, such as ease of navigation between projects and building projects as a group.

Project groups can only contain other projects, but not other project groups. A project group is saved as an XML file with a `.jpgr` file extension and, by default, to the root of the `jbproject` directory. Unlike projects, project groups aren't saved to a project group folder, but only as a file. Projects themselves aren't aware of project groups and can be open standalone and within a project group simultaneously. Changes you make to a project in a group are also made in the standalone project.

### Creating project groups

---

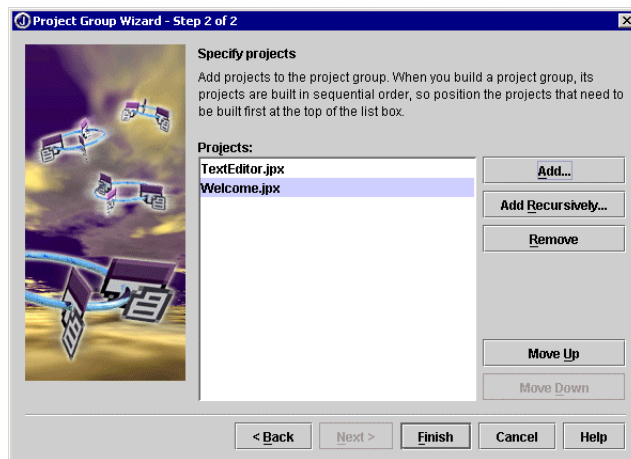
JBuilder provides the Project Group wizard, available on the Project page of the object gallery (File | New), for creating project groups. You can create an empty project group or populate it with projects when completing the wizard. You can add projects to a project group at any time as described in [“Adding and removing projects from project groups” on page 3-3](#). Once you've created a project group, you can also add new projects with the Project wizard. When creating the new project, select the Add Project To Active Project Group option to include it in the project group.

Projects within a project group are built in the order they appear in the project pane. The build order is specified on Step 2 of the Project wizard and can be changed at any time in the Project Group Properties dialog

box. For more information on building project groups, see [“Building project groups” on page 6-5](#).

To create a project group with the Project Group wizard, complete the following steps:

- 1 Choose File | New and click the Project tab of the object gallery.
- 2 Double-click the Project Group icon to open the Project Group wizard.
- 3 Edit the project group file name and/or the location of the file in the File Name field.
- 4 Click Next to continue to the next step.
- 5 Do one of the following:
  - a Choose the Add button, browse to a project, and select it. Click OK to add the project. Repeat to add another project.
  - b Choose the Add Recursively button, select a directory to scan, and click OK. JBuilder scans the selected directory and all its subdirectories and adds all project files (.jpx) to the project group.



- 6 Select a project in the list and use the Move Up or Move Down buttons to reorder the list of projects. Projects are built and displayed in the project pane in the order listed.
- 7 Click OK to close the wizard.

The project group file is displayed at the top of the project pane and the projects are displayed beneath it in the order added. Double-click the project group node to expand and collapse it. Only one project in the group can be active at a time. A project can be open independently and in the project group simultaneously. Double-click a project to make it the active project within the group. Expand the project's node to see its contents. Note that an open, active project is displayed in a bold font in the



project pane and in an italic or bold font in the project drop-down list, depending on the look and feel. For more information about navigating project groups, see [“Navigating project groups” on page 3-4](#).

Once you’ve created a project group, you can close it with File | Close Projects, with the Close button on the project pane toolbar, or by right-clicking the project group node in the project pane and choosing Close Project Group<Name.jpgr>. To open a project group, use File | Open Project or File | Open File. By default, project groups are saved in the `jbpproject` directory, so look for project groups (files with a `.jpgr` extension) in `jbpproject` unless you specified a different location for the project group when you created it with the Project Group wizard.

## Adding and removing projects from project groups

---

You can add projects to and remove projects from a project group at any time. There are several ways to do this:

- Project menu
- Project pane context menu
- Project pane toolbar
- Project Group Properties

To add a project to the open project group, do any of the following:

- Select the project group in the project pane and do one of the following:
  - Choose Project | Add Project.
  - Choose the Add button on the project pane toolbar and browse to the project you want to add.
- Right-click the project group and choose Add Project.
- Choose Project | Project Group Properties, click the Add button, and select the project you want to add.

To remove a project from the open project group, do any of the following:

- Select the project(s) in the project pane and choose Project | Remove From Project Group.
- Right-click the project(s) in the project pane and choose Remove From Project from the context menu.
- Select the project(s) in the project pane and choose the Remove button on the project pane toolbar.
- Choose Project | Project Group Properties, select the project you want to remove, and click the Remove button.

## Navigating project groups

---

One advantage of gathering projects into groups is ease of navigation. Although only one project is active at a time, you can quickly move from one open project to another within the group. You can choose another project from the project pane drop-down list. As with projects, you can also open project groups in multiple AppBrowser windows. Choose **Window | New Browser** to open another AppBrowser window.

## Adding projects as required libraries

---

Projects within project groups often have dependencies upon each other. If you have such a project that is dependent on another, you can add the project upon which your project is dependent to the list of required libraries for your project.

To add a project as a required library,

- 1** Choose **Project | Project Properties** and select the **Paths** page.
- 2** Click the **Required Libraries** tab.
- 3** Click the **Add Project** button.
- 4** Select the project you want to add.
- 5** Click **OK**.

The project you specified is added to the bottom of list of required libraries.

- 6** Click **OK** to close the **Project Properties** dialog box.

Be sure that project you specified as a required library is listed ahead of the project that depends upon it in the project group. That way you know the required project is built first. See [“Building project groups” on page 6-5](#) for information about building project groups.

## Managing paths

Paths are the infrastructure of Java program development. Paths provide a program with what it needs to run. When you set a JDK, you tell the program what path to use to access a JDK. When you create a library, you collate a set of paths that the program will need. Every time a file references another file, it uses a path to get to it.

This section covers how JBuilder constructs paths, how to manipulate paths in the Project Properties dialog box, and how to use path-based tools such as libraries and CodeInsight features.

### Working with libraries

---

JBuilder uses libraries to find everything it needs to run a project as well as for browsing through source, viewing Javadoc, using the visual designer, applying CodeInsight, and compiling code. Libraries are collections of paths that include classes, source files, and documentation files. Libraries are static, not dynamic. Individual library paths are often contained in JAR or ZIP files but can also be contained in directories.

When libraries are added to JBuilder, they are added to the class path so JBuilder can find them. Libraries are searched in the order listed. The order of libraries can be changed in the Configure Libraries dialog box (Tools | Configure Libraries) and on the Paths page of the Project Properties dialog box (Project | Project Properties).

#### See also

- [“How JBuilder constructs paths” on page 4-9](#)

Library configurations are saved in `.library` files and can be saved to several locations:

- **User Home**

Saves the `.library` file to the `<.jbuilder>` directory in the user's home directory.

- **JBuilder**

This is a feature of JBuilder SE and Enterprise

Saves the `.library` file to the `<jbuilder>/lib` directory. Multiple users who are using JBuilder on a network or sharing JBuilder on a single machine have access to the libraries in this folder.

- **Project**

This is a feature of JBuilder SE and Enterprise

Saves the `.library` file in the current project's directory. When using the version control features, the `.library` file is checked in with the other project files.

- **User-defined folder**

This is a feature of JBuilder Enterprise

Saves the `.library` file to a user-defined folder or shared directory. You must add the new folder in the Configure Libraries dialog box before it can appear in the drop-down list.

## Adding and configuring libraries

---

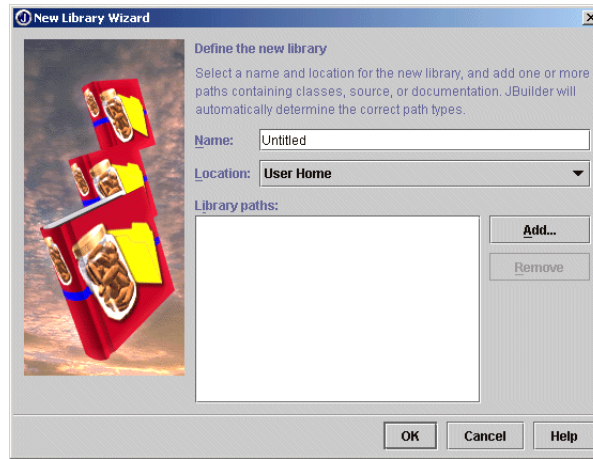
There are several ways to add new libraries to your project. First, you may want to gather your files into JAR files, especially if you plan to deploy your program.

Once you've created the library, add it to JBuilder as follows:

- 1 Select Tools | Configure Libraries. The Configure Libraries dialog box appears.

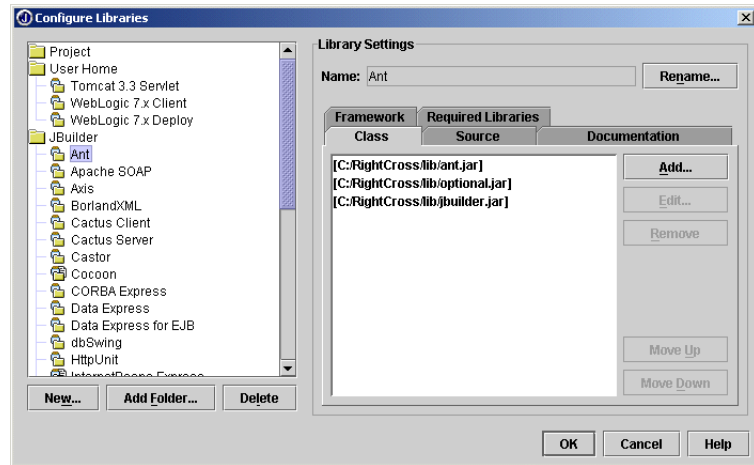
The left-hand pane lets you browse the available libraries. The right-hand pane shows the settings of the selected library.

- 2 Click the New button under the left pane to open the New Library wizard.



- 3 Enter a name for the new library in the Name field.
- 4 Select a location from the drop-down list to save the library configurations: Project, User Home, JBuilder, or user-defined folder. User-defined folders are a feature of Enterprise.
- 5 Click the Add button and select one or more paths containing class, source, and documentation files. JBuilder automatically determines the correct path for the files. Click OK. Notice that the selection appears in the Library Paths list.
- 6 Click OK to close the New Library wizard. Note that the library is saved to the appropriate class, source, and documentation paths in the Configure Libraries dialog box. You can also add, edit, remove, and reorder the library lists in this dialog box. JBuilder Enterprise also includes an Add Folder feature and allows you to add a framework as a

library. For information about adding a framework as a library, see “JSP frameworks” in the *Web Application Developer’s Guide*.



7 Click OK or press *Enter* to close the Configure Libraries dialog box.

To add the library to a project, see [“Setting paths for required libraries” on page 2-22](#).

You can also add libraries in the Project Properties dialog box.

- 1 Select Project | Project Properties.
- 2 Select the Required Libraries tab on the Paths page and click the Add button.
- 3 Click the New button to open the New Library wizard.

### See also

- [“Setting paths for required libraries” on page 2-22](#)
- “Using JAR Files: The Basics” at <http://java.sun.com/docs/books/tutorial/jar/basics/index.html>
- “New Library wizard” in online help

## Editing libraries

---

To edit an existing library,

- 1 Select Tools | Configure Libraries.
- 2 Select the library you want to edit from the list of libraries.
- 3 Select the Class, Source, Documentation, Framework, or Required Libraries tab to choose the library path you want to edit.
- 4 Select the library path and click Edit.
- 5 Browse to a file or user-defined folder in the Select Directory dialog box. Click OK.
- 6 Click Add to browse to a library to add.
- 7 Select a library path and click Remove to remove it.
- 8 Reorder library paths by selecting a library path and clicking Move Up or Move Down.

**Tip** JBuilder searches libraries in the order listed.

- 9 Click OK or press *Enter* to close the Configure Libraries dialog box.

## Adding projects as required libraries

---

Projects can have dependencies upon other projects. If you have such a project that is dependent on another, you can add the project upon which your project is dependent to the list of required libraries for your project.

To add a project as a required library,

- 1 Choose Project | Project Properties and select the Paths page.
- 2 Click the Required Libraries tab.
- 3 Click the Add Project button.
- 4 Select the project you want to add.
- 5 Click OK.

The project you specified is added to the bottom of list of required libraries.

- 6 Click OK to close the Project Properties dialog box.

## Display of library lists

There are three possible colors for libraries listed in JBuilder dialog boxes:

**Table 4.1** Colors in library lists

Color	Description	Troubleshooting
Black	The library is defined correctly.	
Red	The library definition is missing.	This typically means the project refers to a library that is not yet defined. It can also mean that the library definition is faulty: either the library has been defined without any paths or there is more than one library with that name.
Gray	Use of this library requires an upgrade.	You need to upgrade your edition of JBuilder in order to use this library. For example, if you have JBuilder Personal, use of the dbSwing library requires that you upgrade to JBuilder Enterprise.

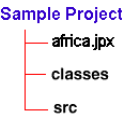
## Packages

Java groups `.java` and `.class` files in a *package*. All the files that make up the source for a Java package are in one subdirectory (`src`) and all compiled files are in another subdirectory (`classes`). When building applications, JBuilder uses the name of the project as the default name for the package in the Application or Applet wizard. For instance, if the project name is `untitled1.jpx`, the Application or Applet wizard suggests using a package name of `untitled1`. Suggested package names are always based on the project name.

Let's look at a sample project to see how the package name affects the file structure.

**Note** In these examples, paths reflect the UNIX platform. See [“Documentation conventions” on page 1-4](#) for information on how paths are documented here.

To organize your project, you might have your project in a folder called `SampleProject`. This project folder contains a project file (`africa.jpx`), a `classes` directory and a `src` directory:



In creating this project, you'll want to create your own packages to hold related sources and classes. In this example, `africa.jpx` contains a package



name of `feline.africa`. This package contains source files on certain felines found in Africa: Lions, Cheetahs and Leopards.

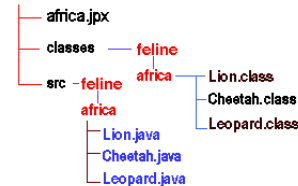
The class files, which are saved in a directory structure that matches the package name, are saved in the `classes` subdirectory within the project. The `src` subdirectory, which contains the `.java` files, has the same structure as the class subdirectory.

#### Sample Project



If the individual classes contained in this project are `Lion.class`, `Cheetah.class`, and `Leopard.class`, these would be found in `classes/feline/africa`. The source files, `Lion.java`, `Cheetah.java`, and `Leopard.java`, would be in found in `src/feline/africa` as shown here.

#### Sample Project



## **.java file location = source path + package path**

It's important to understand what pieces of information JBuilder uses to build the directory location for any given `.java` file. The first part of the directory path is determined by the source path. The source path is defined at the project level and can be modified on the Paths page of the Project Properties dialog box.

Continuing with the `SampleProject` example, the source path for `Lion.java` is:

```
/<home>/<username>/jbproject/SampleProject/src
```

**Note** For the definition of the `<home>` directory, see [“Documentation conventions” on page 1-4](#).

The second part of the directory path is determined by the package name, which in this case is `feline.africa`.

**Note** Java nomenclature uses a period (.) to separate the levels of a package.

The `.java` file location for `Lion.java` is:

```
/<home>/<username>/jbproject/SampleProject/src/feline/africa/Lion.java
```

## See also

- [“How JBuilder constructs paths” on page 4-9](#)

## .class file location = output path + package path

---

The directory location for the .class file is determined by the output path and the package name. The output path is the “root” to which JBuilder will add package paths to create the directory structure for the .class files generated by the compiler. The output path is defined at the project level and can be modified on the Paths page of the Project Properties dialog box.

In the SampleProject example, the output path for Lion.class is:

```
<home>/<username>/jbproject/SampleProject/classes
```

The second part of the directory path is determined by the package name, which in this case is feline.africa.

As shown below, the .class location for Lion.class is:

```
<home>/<username>/jbproject/SampleProject/classes/feline/africa/Lion.class
```

## See also

- [“How JBuilder constructs paths” on page 4-9](#)

## Using packages in JBuilder

---

When referencing classes from a package, you can use an import statement for convenience. An import statement allows you to reference any class in the imported package just by using the short name in the code. (JBuilder’s designers and wizards add import statements automatically.) Here’s an example of an import statement:

```
import feline.africa.*;
```

If this import statement is included in your source code, you could refer to the Lion class as just Lion in the body of the code.

If you don’t import the package, you must reference a particular class in your code with its fully qualified class name. As shown in the following diagram, the fully qualified class name for Lion.java is feline.africa.Lion (package name + class name without the extension).



Packages can be selectively excluded from the build process.

**See also**

- [“Filtering packages” on page 6-23](#)

**Package naming guidelines**

---

The following package naming guidelines are recommended for use in all Java programs. To encourage consistency, readability, and maintainability, package names should be

- One word
- Singular, rather than plural
- All lowercase, even if more than one word (for example, `fourwordpackagename` **not** `FourWordPackageName`)

If your packages will be shared outside your group, package names should start with an Internet domain name with the elements listed in reverse order. For example, if you were using the domain name `foo.domain.com`, your package names should be prefixed with `com.domain.foo`.

**How JBuilder constructs paths**

---

The JBuilder IDE uses several paths during processing:

- Source path
- Output path
- Class path
- Browse path
- Doc path
- Backup path
- Working directory

Paths are set at a project level. To set paths, use the Project Properties dialog box. See [“Setting project properties” on page 2-17](#) for more information.

In the construction of paths, JBuilder eliminates duplicate path names. This prevents potential problems with DOS limitations in Windows.

**Note** In these examples, paths reflect the UNIX platform. See [“Documentation conventions” on page 1-4](#).

## Source path

---

The source path controls where the compiler looks for source files. The source path is constructed from both of the following:

- The path defined on the Source tab of the Paths page of the Project Properties dialog box.
- The directory for generated files. This directory contains source files that are automatically generated by the IDE. Examples of these source files include IDL server and skeleton files. The directory for generated files is placed in the output path. You can change this option on the Build page of the Project Properties dialog box.

The complete source path is composed of these two elements in this order:

source path + output path/Generated Source

Using the `SampleProject` as an example, the source path for the `africa.jpx` project is:

`/<home>/<username>/jbproject/SampleProject/src`

## Output path

---

The output path contains the `.class` files created by JBuilder. The output path is constructed from the path defined in the output path text box, located on the Paths page of the Project Properties dialog box.

Files are placed in a directory whose path matches the output path + the package name. There is only one output path per project.

For example, in the `SampleProject` example, the output path for the `feline.africa.jpx` project is:

`/<home>/<username>/jbproject/SampleProject/classes`

## Class path

---

The class path is used during compiling. This path is constructed from all of the following:

- The output path
- The class path for each library listed on the Paths page of the Project Properties dialog box (in the same order in which they are listed)
- The target JDK version selected on the Paths page of the Project Properties dialog box

The complete class path is composed of these elements in this order:

output path + library class paths (in the order libraries are listed in the Project Properties dialog box) + target JDK version

For example, the complete class path for `Lion.class` is:

```
/<home>/<username>/jbproject/SampleProject/classes:  
/user/jbuilder/lib/dbswing.jar:/
```

The class path is displayed in the message pane when you run the project.

## Browse path

---

The browse path is used by the IDE when you

- Use CodeInsight.
- Choose Find Definition from the editor pop-up menu.
- Choose Search | Find Classes.
- Run the debugger.

The browse path is constructed from all of the following:

- The source path
- The source path for each library listed on the Paths page of the Project Properties dialog box (in the same order in which they are listed)
- The source path for the target JDK version selected on the Paths page of the Project Properties dialog box

The complete browse path is composed of these elements in this order:

source path + library source paths (in the order libraries are listed on the Paths page of the Project Properties dialog box) + JDK target version  
source path

For example, the complete browse path for `Lion.class` is:

```
/<home>/<username>/jbproject/SampleProject/src:  
/user/jbuilder/src/dbswing-src.jar:  
/user/jbuilder/src/dx-src.jar
```

## Doc path

---

The doc path is the path or paths that contain HTML documentation files for API class files. This allows reference documentation to be displayed in the Doc page of the content pane.

The doc path can be set on the Paths page of the Project Properties dialog box. Paths are searched in the order listed.

## Backup path

---

JBuilder uses the backup path to store backup versions of source files. The default backup directory is:

```
/<home>/<username>/jbproject/SampleProject/bak
```

**Important** JSP files, HTML files, and some other text files are not treated as source files. These files are backed up in their original directories.

However, you can include these backups in the backup directory of your project instead. To do so,

- 1 Select Project | Project Properties and look at the Paths page.
- 2 Select the Source page inside the Paths page.
- 3 Click Add in the Source page. This brings up the Select One Or More Directories dialog box.
- 4 Browse to the project backup directory, select it, and click OK.

## Working directory

---

The working directory is the starting directory that JBuilder gives a program when it is launched. Any directory may be configured as the working directory. By default, it has the same name as the project file.

It's generally the parent directory of the source directory. It's the default parent directory of the output, backup, documentation, and library directories.

## Where are my files?

---

Each file in a project is stored with a relative path to the location of the project file. JBuilder uses the source path, test path, class path, browse path, and output path to find and save files.

This list explains the purpose of each type of path:

- The source path controls where the compiler looks for source files.
- The test path serves as the source path when you're using unit testing.
- The class path is used during compiling and at runtime and for certain Enterprise editor features.
- The browse path is used by the IDE when using CodeInsight, Find Definition in the editor, searching, and debugging.
- The output path contains the `.class` files created by JBuilder when you compile your project.

**See also**

- [“How JBuilder constructs paths” on page 4-9](#)
- [“Working with libraries” on page 4-1](#)

## How JBuilder finds files when you drill down

---

When you drill down to explore source code, JBuilder searches for the `.java` files using the browse path. For more information about drilling down, see *“Navigating in the source code”* in *Introducing JBuilder*.

## How JBuilder finds files when you compile

---

When you compile your project, JBuilder uses the following paths:

- class path
- source path
- output path

JBuilder looks in the class path to find the location of the `.class` files, the libraries to use, and the target JDK version to compile against. The compiler compares the `.class` files with their source files, located in the source path, and determines if the `.class` files need to be recompiled to bring them up to date. The resulting `.class` files are placed in the specified output path.

For information about compiling files, see [Chapter 6, “Building Java programs”](#) and [Chapter 5, “Compiling Java programs.”](#)

## How JBuilder finds class files when you run or debug

---

When you run and debug your program, JBuilder uses the class path to locate all classes your program uses.

When you step through code with the debugger, JBuilder uses the browse path to find source files.

For information about debugging files, see [Chapter 8, “Debugging Java programs.”](#)





# Compiling Java programs

A Java compiler reads Java source files and any other source files passed to it and produces the Java program in the form of `.class` files containing bytecodes that are the machine code for the Java Virtual Machine (VM). Compiling produces a separate `.class` file for each class and interface declaration in a source file. When you run the resulting Java program on a particular platform, such as Windows NT, the Java interpreter for that platform runs the bytecodes contained in the `.class` files. For general information about compiling in Java, see the Java Development Kit (JDK) compiler overview, “**javac** - The Java programming language compiler” at <http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html>.

The default compiler for the JBuilder IDE, Borland Make for Java (**bmj**), has full support for the Java language. The JBuilder compiler uses smart dependencies checking, so the compiling/recompiling cycle is faster and more efficient. The dependency checker determines the nature of source code changes and only recompiles the necessary files. For more information, see “[Smart dependencies checking](#)” on page 5-2. To understand how the JBuilder compiler works, see “[Borland Make for Java \(bmj\)](#)” on page B-12.

If you prefer to compile from the command line, JBuilder also provides the following command-line tools in JBuilder SE and Enterprise editions:

- JBuilder **-build** command-line option for building projects
- Borland Make for Java (**bmj**), which uses the dependency checker
- Borland Compiler for Java (**bcj**)

JBuilder also provides the option to change compilers. To take advantage of the many JBuilder features, such as smart dependencies checking, UML, and refactoring, it's recommended that you use Borland Make. However, if you wish to use **javac**, you can change compilers on the Build

page of the Project Properties dialog box (Project | Project Properties). For more information, see [“Setting compiler options” on page 5-6](#).

Compiling is only one phase of the JBuilder build system. Other phases include pre-compile, post-compile, clean, package, and deploy. For more information on these phases and the JBuilder build system, see [Chapter 6, “Building Java programs.”](#)

## Smart dependencies checking

---

The Borland Make compiler provides fast yet complete compiling by using smart dependencies checking, which results in fewer unnecessary compiles of interdependent source files, and thus accelerates the edit/recompile cycle. When compiling, instead of deciding whether to recompile a source file based only on the time stamp of the file, Borland Make analyzes the nature of the changes you make to source files.

There are several possible reasons for recompiling the source:

- One or more of the class files the source would produce are missing.
- The source has been modified since it was last compiled.
- One or more of the classes that the source produces depends on a member in another class that changed.

A change in one source file may change the way other source files are compiled. Not only can JBuilder detect this situation, but it’s capable of noticing when a change in one source would not affect other files, because they refer to portions that haven’t changed. In this case, JBuilder knows **not** to recompile the files.

When you compile source files for the first time, a dependency file is automatically created for each package and is placed in the output directory along with the class files. The dependency file contains detailed information about which class uses which for all the classes in that package. This file has an extension of `.dep2` and is saved in a folder called `package cache` in the same directory as the classes.

Dependency files must be located on the class path so the compiler can find them. When you compile in the IDE, the class path is correctly set by default. For information on how the class path is constructed, see [“Class path” on page 4-10](#).

If you compile from the command line, you might need to set the `CLASSPATH` environment variable. For more information, see [“Setting the CLASSPATH environment variable for command-line tools” on page B-2](#).

Smart dependencies checking is used by Borland Make in the IDE and by the **bmj** command-line make but not by the **bcj** command-line compiler. **bmj** and **bcj** are available in JBuilder SE and Enterprise editions.

**Important** Libraries are considered “stable” and are not checked by the dependency checker.

**See also**

- “JBuilder dependency checker” on Blake Stone’s home page at <http://homepages.borland.com/bstone/articles/depchecker.html>

## Compiling a program

---

The JBuilder IDE uses Borland Make for Java (**bmj**) to compile Java source files. Because the JBuilder compiler uses smart dependencies checking, the compiling/recompiling cycle is faster and more efficient. For more information, see [“Smart dependencies checking” on page 5-2](#). To understand how the JBuilder compiler works, see [“Borland Make for Java \(bmj\)” on page B-12](#).

The following parts of a program can be compiled:

- The entire project
- Packages
- Java files

To understand how JBuilder locates files to compile the program, see [“How JBuilder constructs paths” on page 4-9](#) and [“Where are my files?” on page 4-12](#).

To compile the source files for a program:

- 1 Open the project containing the program or open a single Java file.
- 2 Do one of the following:
  - Choose Project | Make Project.
  - Right-click a Java file(s) in the project pane and choose Make <filename>.
  - Right-click the file tab of an open Java file in the editor and choose Make <filename>.
  - Choose the Make Project button on the toolbar.

## JBuilder build menus

---

JBuilder provides menu commands for building your project: Make, Rebuild, and Clean.

Make is a phase of the JBuilder build system that establishes dependencies among other standalone phases: pre-compile, compile, post-compile,

package, and deploy. The JBuilder Make command is not to be confused with the Java compile make, which only compiles Java source files. The Make command builds Java source files as well as other buildable files in your project, such as archive files, WebApps, and other buildable nodes.

JBuilder also provides the Rebuild command for completely rebuilding your project. The Rebuild command has clean and make as its dependencies. First, all the build output is deleted, then make is executed. Lastly, the Clean command deletes all the build output.

In JBuilder Enterprise, the build menus are configurable. For more information, see [“Configuring the Project menu” on page 6-18](#).

### See also

- [“The Make command” on page 6-3](#)
- [“The Rebuild command” on page 6-4](#)
- [“The Clean command” on page 6-5](#)

## Building projects with the Run command



The Run command can be set to execute a build target before running the project. The default behavior of the Run Project command (Run | Run Project) and the Run Project button is to make and run your application. The default behavior can be changed in the run configurations for the project. For example, you might want to rebuild your project each time before you run it, instead of using the default make. Available build targets vary by the type of project you are working on. Other available targets might include rebuild, clean, none, external build tasks, Ant targets, and any custom build tasks you’ve added in extending the build system through the Open Tools. For information on how to change the build target, see [“Build Targets” on page 7-10](#).

### See also

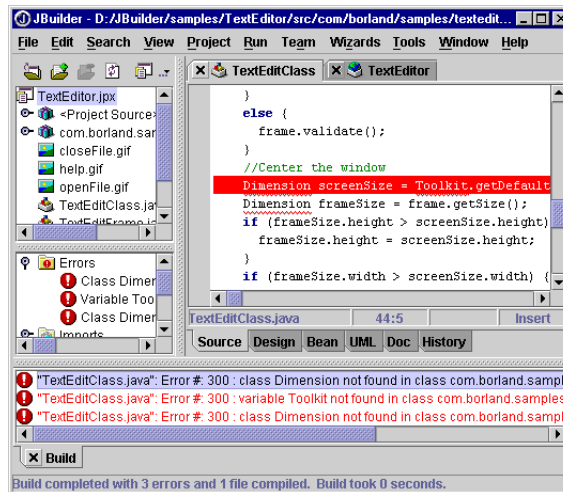
- [“Using the Run command” on page 7-3](#)
- [“Building Ant projects with the Run command” on page 6-12](#)

## Syntax errors and error messages

---

Syntax errors are errors that violate the syntactical rules of the Java programming language. The editor catches these errors as they occur, before you compile. Syntax errors are displayed in a folder at the top of the structure pane. To locate the line of code containing the error, expand the Errors folder in the structure pane and double-click the error. The line containing the error is highlighted in the editor.

Error messages also appear on the Build tab in the message pane during compiling. Select an error message and press *F1* for Help. Use the arrow keys to navigate through compiler error messages. Click an error message to highlight the code in the open file. Double-click an error message to move the cursor to the line of code in the editor.



## See also

- “Error and warning messages” in online help
- “Compiler error messages” in online help, where error messages are listed by number

## Compile problems when opening projects

If you open a project and it won’t compile, check the path settings on the Paths page of the Project Properties dialog box to make sure they are set correctly. JBuilder uses the path settings to construct the class path and source path, which is where the compiler looks for files.

Additionally, check the Required Libraries list on the Paths page. If one or more of the libraries is highlighted in red, it’s not defined for your installation of JBuilder. Double-click the library name or select it and choose Edit to define it. Then, recompile the project.

Projects with WebApps may need a properly configured web server to compile. See “Configuring your web server” in *Web Application Developer’s Guide*.

To set default paths for new projects (to avoid future potential problems), use the Default Project Properties dialog box (Project | Default Project

Properties). See [“Setting project properties” on page 2-17](#) and [“How JBuilder constructs paths” on page 4-9](#).

## Checking for package/directory correspondence

---

JBuilder provides protective checking for duplicate class definitions in a project and for package/directory correspondence. The **bmj** compiler, which is the default compiler in the IDE, verifies that the package statement in a source file corresponds to the package directory and that two source files do not define the same class.

The first time you build a project, all the available `.java` files in a package directory are verified and compiled. If you have temporary sources that you do not want to compile, you should use another extension besides `.java`. For example, if the project contains an old version of a file you are working on and that file contains another definition of the same class, you'll get a “duplicate class definition” error. This checking prevents subtle problems that would be difficult to locate.

## Setting compiler options

---

You can specify compiler options for the current project on the Java page of the Build page of Project Properties (Project | Project Properties). The options, which vary according to the compiler selected, are applied to all files in the project tree. If you change compiler options, you should rebuild your packages or your entire project and not just make them. The project options are applied to any class being rebuilt, outside the project tree as well as inside the project tree.

You can't set compile options per file. However, a file can be used by two projects, both of which have different settings for compiling. Applying options on classes or packages individually is not supported, because there is no separate compilation of headers and modules in Java. If some import information is missing (such as a class file), the imported class is compiled at the same time as the importing class, using the same project-wide options.

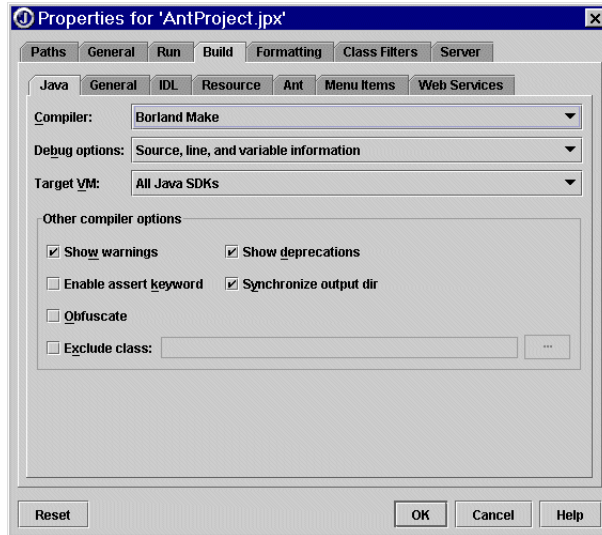
You can also set compiler options for future projects in the Default Project Properties dialog box (Project | Default Project Properties). After setting default project properties, whenever you create a new project with the Project wizard, the default settings are applied.

If you compile your project with Borland Make, compiler options are applied to all files in the project tree and to files referenced by these files, stopping at packages that are marked stable and have no classes in the project tree. Borland Make provides additional compiler options, such as Obfuscate, Synchronize Output Dir, and Exclude Class. For more

information on these options, choose the Help button on the Java page of the Build page.

To set compiler options for your project, complete the following steps:

- 1 Choose Project | Project Properties or right-click the .jpx project node in the project pane and choose Properties. The Project Properties dialog box is displayed.
- 2 Select the Build tab to display the Build page. Then select the Java tab.



- 3 Select a compiler and any debug and compiler options that you want. The available compiler options vary according to the compiler selected. For more information on the options, see the Build page of the Project Properties dialog box (Project | Project Properties). Switching compilers is a feature of JBuilder Enterprise.
- 4 Set any desired options on other pages of the Build page.
- 5 Choose OK to close the Project Properties dialog box and save your settings.
- 6 Choose Project | Rebuild Project to rebuild your project with the revised settings.
- 7 Click OK to close the dialog box.

## Specifying a compiler

---

This is a feature of  
JBuilder Enterprise

By default, JBuilder compiles projects with Borland Make for Java (**bmj**). Generally, it's recommended that you compile with the Borland compiler to take full advantage of the JBuilder features, such as dependencies checking and refactoring. Other available compilers include **javac** and the Project **javac**. When you choose **javac** as the compiler, the project is compiled using the host JDK **javac**, which is located in the JBuilder directory. If Project **javac** is selected, JBuilder uses the JDK's **javac** specified for the project on the Paths page (Project | Project Properties). When you choose a compiler, only options available to that compiler are enabled.

To change the compiler for the project, complete the following steps:

- 1 Choose Project | Project Properties.
- 2 Choose the Build page and click the Java tab.
- 3 Choose a compiler from the Compiler drop-down list.
- 4 Click OK to close the dialog box.

## Setting additional compiler and build options

---

On the Java page of the Build page of Project Properties (Project | Project Properties), you can choose a compiler, debug options, target VM, and other compiler options, such as Show Warnings, Show Deprecations, and Enable Assert Keyword. For more information on these options, choose the Help button on the Java page of the Build page. There are also additional options on the General page of the Build page that affect building.

## Setting the output path

---

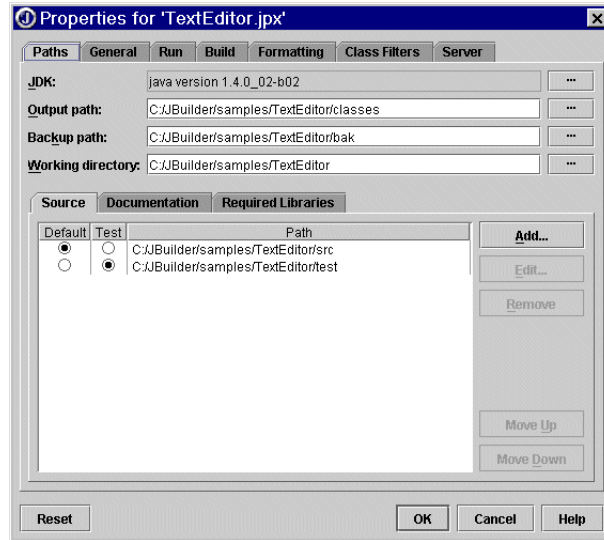
You can set the output path for your compiled class files in the Project Properties dialog box.

To set the output path,

- 1 Choose Project | Project Properties.



## 2 Choose the Paths tab to display the Paths page.



- 3 Choose the ellipsis (...) button to the right of the Output Path field.
- 4 Browse to the directory you want your compiled class files to be saved in and select it. If the directory does not exist, select the New Folder button and create one. Click OK.
- 5 Click OK to close the dialog box.

### See also

- [“How JBuilder constructs paths” on page 4-9](#)
- [“Where are my files?” on page 4-12](#)

## Compiling projects within a project group

This is a feature of  
JBuilder Enterprise

For information on compiling projects within a project group, see [“Building project groups” on page 6-5](#).

## Compiling from the command line

These are features of  
JBuilder SE and  
Enterprise

You can compile from the command line using the `bmj` or `bcj` commands. To see the syntax and list of options, type `bmj` or `bcj` at the command line from the `<jbuilder>/bin` directory. You can also build a project from the JBuilder command line. You might need to set the `CLASSPATH` environment variable for the command line, so that the required classes are found.

## bmj (Borland Make for Java)

---

This is a feature of  
JBuilder SE and  
Enterprise

The **bmj** compiler is the Borland Make for Java. **bmj** compiles any `.java` files that have outdated or nonexistent `.class` files. **bmj** also compiles any imported classes that have outdated or nonexistent `.class` files.

**bmj** looks for dependency files on the class path, and does dependencies checking. If you specify a set of sources, some or all of those sources might not be recompiled. For example, the class files might be determined to be up to date if they have been saved but not edited since the last compile. You can force recompilation using the **-rebuild** option.

To check a set (or “graph”) of interdependent modules, it is sufficient to call **bmj** on the root source (or multiple root sources if one is not under the other). You can specify this argument using source names, package names, class names, or a combination.

### See also

- [“Borland Make for Java \(bmj\)” on page B-12](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page B-2](#)

## bcj (Borland Compiler for Java)

---

This is a feature of  
JBuilder SE and  
Enterprise

The **bcj** compiler is the Borland Compiler for Java. **bcj** compiles the specified sources, whether or not their `.class` files are outdated. It also compiles any directly imported `.java` files that do not have `.class` files. Imported `.java` files that already have `.class` files aren’t recompiled, even if their `.class` files are outdated. After using **bcj**, some imported classes might still have outdated `.class` files.

**bcj** does not do dependencies checking and does not use or generate a dependency file. **bcj** only compiles the items you specify.

### See also

- [“Borland Compiler for Java \(bcj\)” on page B-7](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page B-2](#)

## Building a project from the command line

---

This is an option in  
JBuilder SE and  
Enterprise

You can build project files and specify build targets from the JBuilder command line using the **-build** option from the `<jbuilder>/bin` directory. For more information, see [“JBuilder command-line interface” on page B-4](#).

## Switching between the command line and IDE

---

If you edit a file outside the IDE, be sure to include its package or at least one of that package's sources in your project, so that the package is checked when you compile. Otherwise, the change isn't detected and the source isn't recompiled.



# Building Java programs

Build features vary by  
JBuilder edition

JBuilder's build system, based on the Java-based build tool Ant, involves various build phases. Build phases, which are special targets that the build system always creates for every build process, can include such build tasks as preparing non-Java files for compiling, compiling Java source files, archiving, deploying, and so on. The build system can be customized and extended with the `OpenToolBuilder` class.

The JBuilder compiler, Borland Make for Java (**bmj**), has full support for the Java language, including inner classes and JAR files. Because the JBuilder compiler uses smart dependencies checking, the compiling/recompiling cycle is faster and more efficient. The dependency checker determines the nature of the changes and recompiles only the necessary files.

## See also

- [“Smart dependencies checking” on page 5-2](#)
- [“Compiling Java programs” on page 5-1](#)
- [“Borland Make for Java \(bmj\)” on page B-12](#)

## The JBuilder build system

---

JBuilder's build system uses Ant, an open source, Java-based build tool, programmatically to execute builds as opposed to using static Ant build files. The build system, also extensible as an `OpenTool`, has several advantages and allows you to do the following:

- Extend the build system with an `OpenTool` and track the output.
- Specify dependencies between build targets, a feature of JBuilder Enterprise.

- Build project groups, a feature of JBuilder Enterprise.
- Build existing Ant projects in JBuilder, a feature of JBuilder Enterprise.
- Filter packages and remove them from the build process, a feature of JBuilder SE and Enterprise.

### See also

- “JBuilder Build System Concepts” in the OpenTools online Help
- [“Building project groups” on page 6-5](#)
- [“Building with external Ant files” on page 6-7](#)
- [“Filtering packages” on page 6-23](#)

## Build system terms

---

The following terms are used in discussing the build system.

**Table 6.1** Build system terms

Term	Definition
Build task	A piece of code that can be executed during the build process, such as java compilation, FTP, creating a JAR file, and so on.
Target	Collections of zero or more build tasks to be executed. Targets can have dependencies on other targets. For example, if target A depends on targets B and C, B and C are executed before A is executed.
Phase	Special targets that the JBuilder build system always creates for every build process. There are eight phases: six standalone phases without dependencies (clean, pre-compile, compile, post-compile, package, and deploy) and two phases that establish dependencies among phases (make and rebuild). Each phase has its own specific targets.

---

## Build phases

---

The build phases in the JBuilder IDE include six standalone phases without dependencies and two phases that establish dependencies among the other phases. Every JBuilder project has the following standalone phases: clean, pre-compile, compile, post-compile, package, and deploy. Because each phase is standalone, a phase can be executed without needing to execute any other phase. Each phase has its own targets as dependencies. For example, SQLJ is a dependency of the Post-compile phase.

Two additional phases establish dependencies among the six standalone phases: make and rebuild. Make has these dependencies in the order listed: pre-compile, compile, post-compile, package, and deploy. Rebuild has clean and make as dependencies.

Because the JBuilder build system is exposed as an OpenTool, you can create your own build tasks and specify existing phases as dependencies or not tie into the existing phases at all. See “JBuilder build system concepts” in the OpenTools online Help for more information on extending the build system. See the Obfuscator sample in the JBuilder `samples/opentoolsAPI/Build` directory for a sample on creating Builders.

**Table 6.2** Build system phases

<b>Standalone phases</b>	
<b>Term</b>	<b>Definition</b>
Clean	Removes all build output, such as .class files, JARs, and so on.
Pre-compile	Tasks that occur before compiling. IDL files, which are converted to Java source files before compiling, are examples of a Pre-compile target.
Compile	Generation of Java class files from Java source files.
Post-compile	Tasks that occur after compiling. This phase requires Java class files to be executed. java2iio and obfuscated code could be targets of this phase.
Package	Tasks that generate archive files.
Deploy	Tasks that move deployed files to another location. For example, this phase might have a task to FTP files.
<b>Phases that establish dependencies</b>	
<b>Term</b>	<b>Definition</b>
Make	Make establishes dependencies among the standalone phases in this order: pre-compile, compile, post-compile, package, and deploy.
Rebuild	Rebuild has clean and make as dependencies.

## The Make command

Make is a phase that establishes dependencies among the standalone phases. Make has the following dependencies in the order listed: pre-compile, compile, post-compile, package, and deploy.

The Make command is not to be confused with the Java compile make, which only compiles Java source files. For more information about the JBuilder compiler, see [“Borland Make for Java \(bmj\)” on page B-12](#) and [“Smart dependencies checking” on page 5-2](#).

Make executes various build tasks, depending on the nodes selected. The selected nodes can be a project, packages, Java source files, or other appropriate nodes, such as archive, documentation, WebApp, external build task, or Ant target nodes. For example, if you make an archive node, an archive file is generated. When you make a package, Java source files are compiled and resources in these packages are copied to the project’s output path. Making a project compiles the Java source files in the project, as well as executing the appropriate build tasks on any buildable nodes.

There are several ways to make a file, project, package, or other appropriate node:

- Choose Project | Make Project.
- Choose Project | Make <filename>.
- Choose the Make Project button on the toolbar, if available.
- Right-click a node in the project pane and choose Make.
- Right-click the file tab in the content pane and select Make <filename>.

In addition, in JBuilder Enterprise, you can make a project group. For more information on project groups, see [Chapter 3, “Working with project groups.”](#)

## The Rebuild command

Rebuild is another phase that establishes dependencies among the standalone phases. It has clean and make as dependencies. Rebuild deletes all the build output with clean, then does a make. The selected node can be anything that’s buildable that supports clean. Some examples include projects, packages, Java source files, archives, and resources.

Because Rebuild executes clean and then make, it takes longer than make. But it’s useful if you want a clean build. For example, if you’ve deleted Java source files, you would use Rebuild. Rebuild executes Clean, which removes all of the build output, including the class files. Then make is executed. If you were to do a make after deleting the Java source files, their class files would still exist.

**Important** If you change any debug or obfuscation options on the Build page of Project Properties, you must rebuild your project for these changes to take effect.

There are several ways to rebuild a file, project, package, or other appropriate node:

- Choose Project | Rebuild Project.
- Choose Project | Rebuild <filename>.
- Right-click a node in the project pane and choose Rebuild.
- Right-click the file tab in the content pane and select Rebuild <filename>.
- Choose the drop-down list next to the Make button on the toolbar and choose Rebuild Project.

In addition, in JBuilder Enterprise, you can rebuild a project group. For more information on building project groups, see [“Building project groups” on page 6-5.](#)



## The Clean command

The Clean command removes all build output of the other targets, such as the `classes` directory, JARs, WARs, and so on. If the source and output paths are the same, the output directory is not deleted but the build output is deleted. What Clean removes is dependent upon the node selected:

- Project nodes: recursively deletes the output directory. This only occurs if the output directory is a subdirectory of the project. Clean doesn't delete the output directory if it's the same as the source directory or a subdirectory of the source directory.
- Java nodes: deletes the corresponding `.class` files and any generated files, such as `java2iop`. Also removes resources.
- Package nodes: deletes the corresponding `.class` files and any resources.
- Resource nodes: deletes the copies in the output directory.
- Documentation nodes: deletes all HTML and HTM files in the Javadoc output directory.
- Archive nodes: deletes the archive file(s) and executables.
- WebApp nodes: deletes any WAR files and the `WEB-INF/lib` and `WEB-INF/classes` directories.

There are several ways to Clean:

- Right-click the project file in the project pane and choose Clean.
- Right-click an appropriate node or nodes in the project pane and choose Clean.

As with the Make and Rebuild commands, the Clean command only appears on the context menu when appropriate nodes are selected.

## Building project groups

---

This is a feature of  
JBuilder Enterprise

Using project groups allow you to control the build order of the projects within the group. This is particularly useful if one project is dependent on another. In this case, you would want to build the dependency first. For example, if project B is dependent on project A, you would build project A first, then project B.

The build order of projects within a project group can be modified on the Build Order page of the Project Group Properties dialog box (Project | Project Group Properties).

### See also

- [Chapter 3, “Working with project groups”](#)

## Specifying the build order for a project group

---

The build order of a project group is determined by the order of the projects in the project pane. For example, if a project group has two project nodes, `project1.jpx` and `project2.jpx`, and `project1.jpx` is the first child node of the project group, then JBuilder builds `project1.jpx` first and `project2.jpx` last. The build order can be changed in the Project Group Properties dialog box.

Controlling the build order in a project group can be especially useful if a project has another project added as a required library. If you want the required project built first, then you need to put both projects in a project group and have the required project first in the project group. For more information, see [“Adding projects as required libraries” on page 3-4](#).

To change the build order in a project group,

- 1 Open Project Group Properties:
  - Choose Project | Project Group Properties.
  - Right-click the project group node in the project pane and choose Properties.
- 2 Click the Build Order tab on the Build page.
- 3 Select a project in the list and use the Move Up or Move Down buttons to reorder the build order.

**Tip** You can also add projects in the Project Group Properties dialog box. Choose the Help button for more information.

- 4 Click OK to close the dialog box. Notice that the order of projects in the project pane changes according to the new build order you just specified.

## Building a project group

---

To build or rebuild a project group,

- 1 Specify the build order of the subprojects in the group as described in [“Specifying the build order for a project group” on page 6-6](#).
- 2 Do one of the following:
  - Choose Project | Make Project Group or Project | Rebuild Project Group.
  - Right-click the project group file (`.jpgx`) in the project pane and choose Make Project Group or Rebuild Project Group.



Make Project Group and Rebuild Project Group are also on the toolbar. By default, Make Project Group is the button on the toolbar and Rebuild Project Group is on the drop-down list next to the button. If you add any custom targets, they're also displayed on the drop-down list. The first two menu choices of the project group build portion of the Project menu are assigned keyboard mappings.

### See also

- [“The Make command” on page 6-3](#)
- [“The Rebuild command” on page 6-4](#)

## Adding project group build targets to the Project menu

---



JBuilder allows you to add new targets to the Project menu for project groups and to customize the menu order. You can add a Clean Project Group menu command, as well as custom targets that specify a collection of build targets to execute with one menu command. Any targets that are added to the Project menu also display on the context menu. For more information, see [“Configuring the Project menu for project groups” on page 6-19](#).

## Building with external Ant files

---

This is a feature of  
JBuilder Enterprise

If you have an existing project that already uses Ant, you can run Ant in JBuilder. Ant is a Java-based build tool that uses build files written in XML. The build files use a target tree where various tasks are executed. A target, which is a set of tasks to be executed, can depend on other targets. Examples of targets include compiling, packaging into JARS for distribution, cleaning directories, and so on.

For example, the following build file example has two targets, `init` and `compile`. The `init` target executes a task that creates a `build` directory. The `compile` target, which depends on the `init` target, executes the `javac` task on the `src` directory and sends the compiled classes to the `build` directory. Because `compile` is dependent on `init`, `init` must execute first. The `build` directory must be created before the classes can be compiled. Build files also have a default target, `compile` in this example, which is executed if a target isn't specified.

Build files can have a set of properties that have a case-sensitive name and a value. Properties can be used as values in tasks and are surrounded by `${ }`. In this example, the property `build` has a value of `build`. The `init` target, when it executes the `<mkdir dir="${build}"/>` task, creates a `build` directory according to the property value, `build`.

## Build file example

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="MyProject" default="compile" basedir=". ">
  <!--
    [ property definitions ]
    [ path and patternset ]
    [ targets ]
  -->
  <property name="build" value="build"/>
  <property name="src" value="src"/>
  <target name="init">
    <mkdir dir="${build}"/>
  </target>
  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>
</project>
```

For more about build files, see the Ant documentation in the JBuilder extras/Ant/docs/ **directory** or at <http://jakarta.apache.org/ant/manual/using.html#buildfile>.

## See also

- [Chapter 18, “Tutorial: Building with Ant files”](#)
- The Jakarta project at Apache: <http://jakarta.apache.org/ant>
- Ant documentation at <http://jakarta.apache.org/ant/manual/index.html>
- Ant documentation in the JBuilder extras/ant/docs/ **directory**

## Adding Ant build files to projects

---

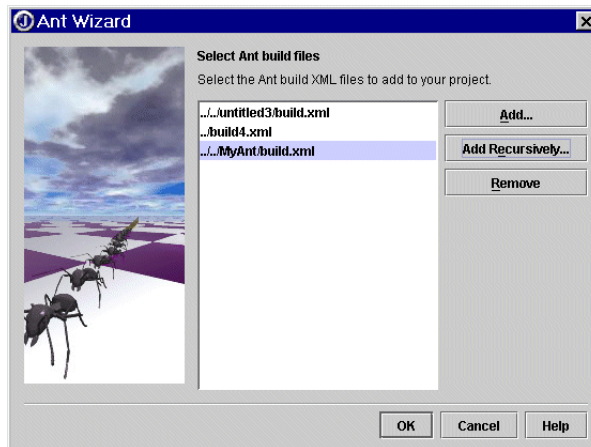
There are two ways to add Ant build files to a project: automatically with the Ant wizard or manually with Project | Add Files/Packages. If you add build files with the Ant wizard, JBuilder automatically recognizes them as Ant nodes and displays Ant icons for the build file nodes. If you add your build files manually with Project | Add Files/Packages, build files named `build.xml` are the only files recognized as Ant build files. You can use other names for the build files, but you must set an option in the node properties for JBuilder to recognize them as Ant files. Also, when the Ant build file is named `build.xml`, the relative path to the file displays in the project pane. For more information on changing the properties for Ant nodes, see [“Setting Ant properties” on page 6-12](#).

## Adding Ant files with the Ant wizard

The easiest way to add Ant build files to your project is with the Ant wizard. The wizard automatically sets the property for the file to an Ant build file, so JBuilder recognizes it as an Ant node, regardless of the file name. Once you've added a build file to the project with the wizard, it's displayed in the project pane with an Ant icon.

To add an Ant build file with the wizard,

- 1 Choose File | New, click the Build tab of the object gallery, and double-click the Ant icon or choose Wizards | Ant.
- 2 Do one of the following:
  - Choose the Add button, browse to any build XML files that you want to add, and click OK. When you use the Add button, any XML file that you add automatically has its property set to an Ant build file so JBuilder recognizes it as an Ant node, regardless of the file name. Once you've added a build file with the Add button, it displays in the project pane with an Ant icon.
  - Choose the Add Recursively button, select a directory, and click OK. JBuilder scans all files named `build*.xml` in the selected directory and all its subdirectories and adds them to the project.



- 3 Click OK to close the wizard.

## Adding Ant files manually

If you manually add Ant build files to your project, they must be named `build.xml` for JBuilder to recognize them automatically. If a file has a different name, it displays in the project pane with the usual XML icon. For JBuilder to recognize it as an Ant file, you must set the node properties as described in the last step here.

To add an Ant build file manually,

- 1 Choose the Add Files/Packages button on the project pane toolbar or choose Project | Add Files/Packages.
- 2 Browse to and select the build file you want to add.
- 3 Click OK.

**Note** If the Ant build file isn't named `build.xml`, change the node properties for the file. Right-click the build file in the project pane and choose Properties. Choose the Ant tab on the Properties page and select the Ant Build File option. Click OK to close the Properties page. The build file displays with an Ant icon.

## Creating and editing Ant build files

---

If you don't have an existing build file, you can create one in the JBuilder editor, which also provides syntax highlighting. You can also use the JBuilder editor to edit any existing Ant build files. To create a new build file in JBuilder,

- 1 Open a project or create a new one.
- 2 Choose File | New File.
- 3 Enter a file name in the Name field, choose XML as the file extension from the Type drop-down list, specify a directory for the file, and click OK. If you name the file `build.xml`, it's automatically recognized as an Ant build file. If the file isn't named `build.xml`, you need to set the Ant Build File option in the Ant properties, so JBuilder will recognize it as an Ant node. See ["Setting Ant properties" on page 6-12](#).
- 4 Enter text in the new file open in the editor.
- 5 Choose Project | Add Files/Packages, select the new file, and click OK to add it to your project.
- 6 Click OK to save the new file to your project.
- 7 Input the appropriate build information in the new file in the editor and save the file.
- 8 Click the refresh button and expand the Ant node in the project pane to display the targets.

## Importing existing Ant projects

---

If you already have an existing Ant project that you'd like to work with in JBuilder, use the Project For Existing Code wizard. The Project For Existing code wizard creates a new JBuilder project from an existing body of work, deriving paths from the directory tree. JBuilder automatically

recognizes Ant `build.xml` files and adds them to your new JBuilder project. If the Ant build file isn't named `build.xml`, you need to set the Ant Build File option on the Ant properties page. See [“Setting Ant properties” on page 6-12](#). To open the Project For Existing Code wizard, choose File | New, click the Project tab, and double-click the Project For Existing Code icon.

## Building Ant projects

---

When you work with an Ant project, you can run Ant as part of the JBuilder build process. To do this, you must set the Always Run Ant When Building Project option for any Ant nodes that you want to include in the build process. See [“Setting Ant properties” on page 6-12](#). Once you've set this option, the Make Project and Rebuild Project commands run Ant as part of the JBuilder build process. If this option is off, the Make Project and Rebuild Project commands run the JBuilder build process without running Ant. See [“Building Ant projects with the Run command” on page 6-12](#).

**Tip** You can add Ant targets to the Project menu and the toolbar. See [“Configuring the Project menu” on page 6-18](#).

You can also run Ant manually from the project pane. Simply right-click the Ant node and choose Make to run the default target in the Ant build file. The default target is a value specified in the `<project>` element. To run several targets in the build file, select one or more of the target nodes, right-click, and choose Make.

**Note** JBuilder might use different paths and directories for source files, class files, and other files. You can change the JBuilder paths to match your Ant targets on the Paths page of Project Properties. You can also change the Ant paths by changing the Ant properties. See [“Setting Ant properties” on page 6-12](#).

Output from Ant displays on the Build tab of the message pane. Two nodes can display messages:

- `StdErr`: displays the standard error output stream.
- `StdOut`: displays the standard output stream.

To navigate to files with errors in them, click the error messages in the message pane. Double-click an error to move the cursor to the error in the code.

If you want to pass different target parameters but not modify the build file, right-click the Ant node, choose Properties, and add the parameters. See [“Setting Ant properties” on page 6-12](#).

## Specifying the JDK

By default, Ant uses the JDK shipped with JBuilder to build projects. In some cases, you might have your project using a different JDK. If you want Ant to use the same JDK as your project, you can set the Use Project JDK When Running Ant option in the Project Properties.

To set Ant to use the project's JDK,

- 1 Choose Project | Project Properties.
- 2 Click the Build tab, then the Ant tab.
- 3 Check the option, Use Project JDK When Running Ant.
- 4 Click OK to close the dialog box.

## Building Ant projects with the Run command

When you run an Ant project in JBuilder with the Run Project command (Run | Run Project), JBuilder runs the default build target, Make. Then, JBuilder runs the project without running Ant. If you want to also run Ant with this command, you must set the Always Run Ant When Building Project option for any Ant nodes that you want to include in the build process. Then JBuilder runs make for the JBuilder build process and Ant, using the default Ant target in the build file. See [“Setting Ant properties” on page 6-12](#). Once this option is set, choosing the Run Project command builds the project with Ant on the specified nodes as part of the JBuilder build process, then runs the program.

In addition, you can change the default Make build target that executes before running the program. For example, you might want to execute an Ant target before running the project. The Always Run Ant When Building Project option doesn't need to be selected in this case. The build target is specified in the runtime configurations in the Runtime Properties dialog box (Run | Configurations). For information on how to change the build target, see [“Build Targets” on page 7-10](#).

### See also

- [“Building projects with the Run command” on page 5-4](#)
- [“Using the Run command” on page 7-3](#)

## Setting Ant properties

---

An Ant project can have a set of properties. Many Ant targets and tasks are typically “property-aware.” For example, there is a property, `build.compiler`, that specifies which compiler the `javac` task uses. You can also specify whether a task is executed based on the existence or non-existence of a property. Properties are also the mechanism used to



pass parameters to tasks without overriding the existing properties in the build file.

You can pass parameters and control the options for launching Ant in the Properties dialog box for the build file. Right-click the Ant node in the project pane, choose Properties, and click the Ant tab. In the Properties dialog box, you can set such options as:

- **Ant Build File**

Select this option to identify the XML file as an Ant build file. If a build file is named `build.xml`, this option is selected by default and disabled.

- **Show Relative Path**

Select this option to display the relative path in the project pane for any Ant build file named `build.xml`.

- **Log Level**

Choose quiet, normal, verbose, or debug for message output.

- **Use Log File**

Output messages to a log file instead of the message pane.

- **Properties**

Add new properties and modify existing properties without overwriting them in the build file. Modifications are saved in the `<project>.jpx` file, not in the XML file.

- **Use Borland Java Compiler**

Use Borland Java Compiler (**bmj**) for **javac** tasks.

- **VM Parameters**

Specify any additional VM parameters when running Ant as an external process from within JBuilder. For example, if you want the maximum heap of the Java VM in which you are running Ant to be 256MB, enter `-Xmx256m` in the VM Parameters field.

- **Task Scheduling:**

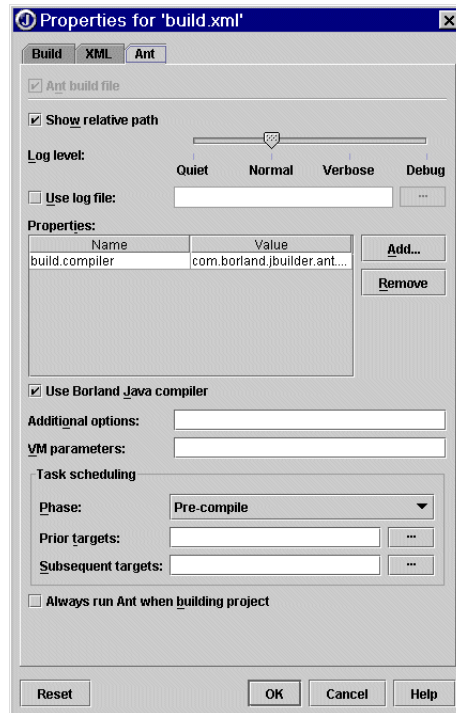
Specify the build phase in which the build file is executed and choose the targets that execute before and after the Ant file is built.

- **Always Run Ant When Building Project**

Run Ant on any Ant node that has this option selected when building the project.

**Important** When you're using JBuilder features, such as refactoring, it's recommended that you accept the default option Use Borland Java Compiler. When it's selected and you have any **javac** tasks in your `build.xml` file, those tasks will use **bmj**. For example, **bmj** puts additional information in its dependency files that allows refactoring to work for

certain edge cases, where the necessary information for refactoring cannot be gleaned from .class files alone.



For more information on setting Ant properties, choose the Help button on the Ant properties dialog box.

## Ant options

You can enter additional Ant options in the Additional Options field of the Properties dialog box.

Options include:

<code>-help</code>	print this message
<code>-projecthelp</code>	print project help information
<code>-version</code>	print the version information and exit
<code>-emacs</code>	produce logging information without adornments
<code>-logger classname</code>	the class that is to perform logging
<code>-listener classname</code>	add an instance of class as a project listener
<code>-find file</code>	search for buildfile towards the root of the filesystem and use the first one found

## Adding custom Ant libraries

If your Ant targets require any Ant libraries, you can add them to your project on the Build page of the Project Properties dialog box.

- 1 Choose Project | Project Properties and click the Build tab.
- 2 Choose the Ant tab.
- 3 Add libraries as needed and click OK.
- 4 Reorder the libraries in the list using the Move Up and Move Down button.

**Note** The libraries are searched in the order listed.

- 5 Click OK to close the dialog box.

You can also use a different version of Ant by adding a library with the Ant JARs. If you don't specify any Ant JARs, JBuilder uses the Ant delivered in the JBuilder `lib` directory.

## Building SQLJ files

---

This is a feature of  
JBuilder Enterprise

The JBuilder build system provides support for building SQLJ files. SQLJ combines the Java programming language with SQL (Structured Query Language), which is used to access relational databases. SQLJ, complementary to JDBC, allows a Java program to access a database using embedded SQL statements. Once the SQL statements are embedded, a SQLJ translator is run on the program. The translator converts the SQLJ program to Java and replaces the SQL statements with calls to the SQLJ runtime. Then, the Java program is compiled and run against the database. While SQLJ supports only static SQL, you can use it in combination with JDBC in an application to also work with dynamic SQL.

For more information on SQL and databases, see the *Database Application Developer's Guide*.

JBuilder recognizes `.sqlj` files in the build process. To generate `.sqlj` files, you must first configure a translator and then specify one for your project as follows:

- 1 Configure DB2 or Oracle SQLJ in the Enterprise Setup dialog box as follows:
  - a Choose Tools | Enterprise Setup and choose the SQLJ tab.
  - b Select a SQLJ configuration to set up, such as Oracle or DB2.
  - c Browse to the location of the executable file.
  - d Enter any additional options.
  - e Add any SQLJ-dependent libraries and JDBC drivers required by the SQLJ translator. Check the DB2 or Oracle documentation to see which JARs are need on your CLASSPATH. Create a library of these JARs with the New Library wizard and add it to your SQLJ setup.
  - f Click OK to close the Enterprise Setup dialog box.

## 2 Specify the SQLJ translator to use for your project:

- a Choose Project | Project Properties and choose the Build tab.
- b Choose the General tab on the Build page.
- c Choose a SQLJ translator for the project.
- d Click OK to close the Project Properties dialog box.

Once your project has an active SQLJ translator, SQLJ is run against any .sqlj files in your project as part of the build process, and the generated .java files appear as children of the SQLJ node. The generated .java files are then compiled as part of the overall build process.

### See also

- SQLJ.org at <http://www.sqlj.org>

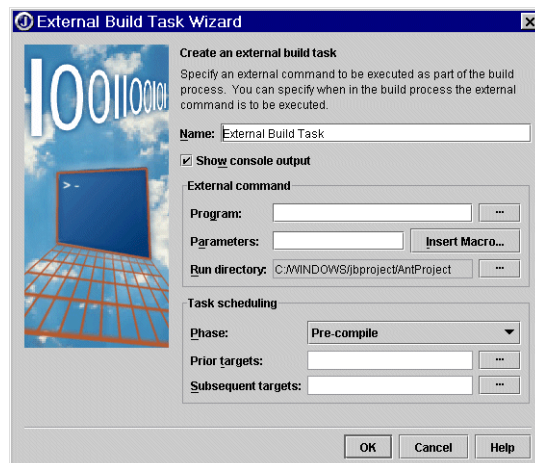
## Creating external build tasks

This is a feature of  
JBuilder Enterprise

There may be cases where you want to execute external tasks whenever you build a project. For example, you might have a .bat or .exe on Windows or a .sh or executable on Linux or UNIX that you want to execute every time you do a build. Examples of external tasks are: passing compiled files to an obfuscator after the Package phase, deploying a program and checking it into CVS, or passing a program to a preverifier after compiling. With the External Build Task wizard, you can create external tasks that allow you to execute external shell or console commands as part of the build process.

### External Build Task wizard

You can create an external build task with the External Build Task wizard. To open the wizard, choose File | New, choose the Build page, and double-click the External Build Task icon.



In the External Build Task wizard, you can set the following options:

- **Name**

Enter a name for the node that displays in the project pane.

- **Show Console Output**

Displays any console output in the JBuilder message pane. If this option is off, no output is displayed.

- **Program**

Browse to the external program file you want to include in the build.

- **Parameters**

Enter any parameters and/or choose from the Macros List.

- **Run Directory**

Specify the directory that your external build task launches from.

- **Task scheduling**

Specify the build phase in which the task is executed and choose the targets that execute before and after the external task.

Once you've completed the wizard, a node displays in the project pane. You can have multiple external task nodes in a project. Each node displays a tool tip with the executable name when you position the mouse over it.

External build tasks can be added to the Project menu. See [“Configuring the Project menu” on page 6-18](#). They can also be specified as the build target to execute before running a project. See [“Build Targets” on page 7-10](#).

## Building external tasks

---

To build only the external task node, right-click it in the project pane and choose Make. If the Show Console Output option is selected, messages are routed to the Build tab in the message pane. Two nodes can display messages:

- StdErr: displays the standard error output stream.
- StdOut: displays the standard output stream.

Choose Project | Make Project to build the entire project and the external task node.

## Setting external build task properties

---

An external build task has a set of properties that are set initially in the External Build Task wizard. These properties include name, executable to

be executed, task scheduling, and parameters. Task scheduling determines the phase in which the task is executed. For example, if your external build task is an obfuscator, you would set the task in the Package phase. You can also specify the targets that occur before and after the task within the specified phase.

To modify any of these properties after you've created the external build task, right-click the node in the project pane and choose Properties.

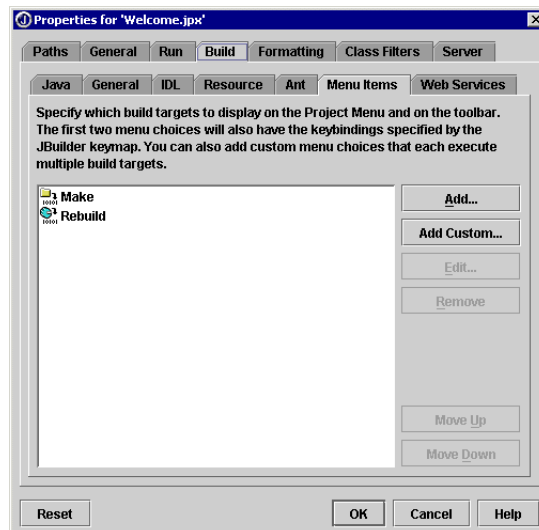
## Configuring the Project menu

This is a feature of  
JBuilder Enterprise

For convenience, JBuilder allows you to configure the first group of the Project menu. By default, Make and Rebuild are the targets on the Project menu. You can add additional targets and build tasks, such as Clean, external build tasks, and Ant targets if your project contains them.

The first two menu items on the Project menu are assigned default key bindings. The first menu item also displays on the toolbar. Next to the toolbar menu button is a drop-down menu that contains Rebuild, unless you've removed it from the menu, and any other custom targets that you've added to the Project menu.

You can change these menu defaults, as well as add custom targets, on the Menu Items page of the Build page in Project Properties. The first two targets listed on the Menu Items page have configurable key bindings. You can change the key bindings for these first two targets in the Keymap Editor found on the Browser page of the IDE Options dialog box (Tools | IDE Options | Browser | Customize). The first target in the list displays with the appropriate icon on the main toolbar with a drop-down list of all other targets added to the Project menu.



To add a project-level target to the Project menu,

- 1 Choose Project | Project Properties and click the Build tab.
- 2 Click the Menu Items tab.
- 3 Do one of the following:
  - Click the Add button and choose an available target from the list. Then Click OK to close the Add Build Target To Menu dialog box.
  - Click the Add Custom button to add a custom menu choice that executes multiple build targets. Enter a menu name in the Menu Label field, click the Add button, and select the desired targets. The targets are executed in the order listed. Use the Move Up or Move Down buttons to reorder the list. Click OK to close the New Custom Target dialog box.
- 4 Change the order of the targets on the Menu Items page by selecting a target and choosing Move Up or Move Down. This changes the order of the targets on the Project Menu. As noted previously, the first target in the list appears on the toolbar and the first two targets have configurable key bindings.
- 5 Click OK to close Project Properties. The new targets now appear on the Project menu and on the drop-down menu next to the target on the toolbar.

**Note** To configure the Project menu for future projects, make the changes in the Default Project Properties dialog box.

### See also

- [“Creating external build tasks” on page 6-16](#)
- “Keymaps of Editor Emulations” in *Introducing JBuilder*

## Configuring the Project menu for project groups

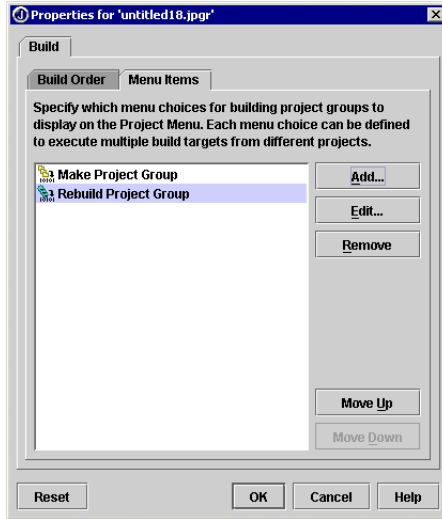
---

This is a feature of  
JBuilder Enterprise

JBuilder also allows you to configure the Project menu for project groups. You can add Clean Project Group to the Project menu, as well as targets that contain collections of targets contained in the subprojects. By default, Make Project Group and Rebuild Project Group are the project group build targets on the Project menu. Any targets that are added to the Project menu also display on the context menu.

The first two project group menu items on the Project menu are assigned default key bindings. The first project group menu item also displays on the toolbar. Next to the toolbar menu button is a drop-down menu that contains Rebuild Project Group, unless you’ve removed it from the menu, and any other custom targets that you’ve added to the Project menu for project groups.

You can change these menu defaults, as well as add custom targets, on the Menu Items page of the Build page in Project Group Properties. The first two targets listed on the Menu Items page have configurable key bindings. You can change the key bindings for these first two targets in the Keymap Editor found on the Browser page of the IDE Options dialog box (Tools | IDE Options | Browser | Customize). The first target in the list displays with the appropriate icon on the main toolbar with a drop-down list of all other targets added to the Project menu.

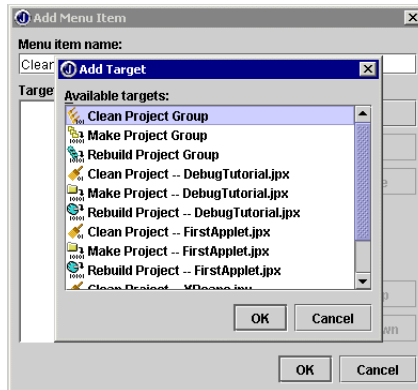


To add a project group target to the Project menu,

- 1 Choose Project | Project Group Properties to open the Project Group Properties dialog box. You can also right-click the project group node in the project pane and choose Properties.
- 2 Choose the Build tab and click the Menu Items tab.
- 3 Click the Add button to open the Add Menu Item dialog box.
- 4 Enter a menu name for the target.
- 5 Click the Add button, select the targets you want to add, and click OK.
- 6 Select a target in the list and use the Move Up or Move Down buttons in the Add Menu Item dialog box to reorder the target in the list. The targets are executed in the order listed.



- 7 Click OK to close the Add Target dialog box.



- 8 Select a menu item in the list and use the Move Up or Move Down button on the Menu Items page to change the order of the target on the Project menu.
- 9 Click OK to close the Properties dialog box.

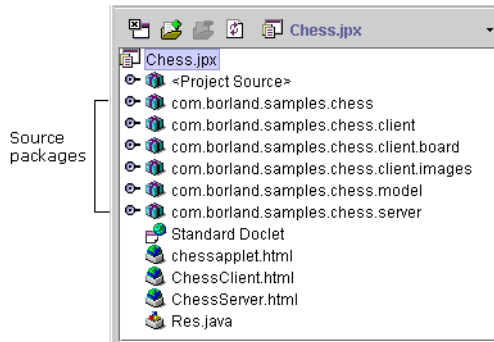
The new target now displays on the Project menu with the other project group build targets.

## Automatic source packages

This is a feature of  
JBuilder SE and  
Enterprise

When the automatic source packages feature is enabled, all packages in the project's source paths automatically appear in the project pane. When building with this option on, any packages that contain buildable files are automatically built and copied with any resources to the project's output path. For example, if a project contains Java and SQLJ files, the Java files are compiled and SQLJ is run against any SQLJ files. By default, JBuilder considers resources to be images, sound, and properties files. Resources can be defined on individual files and by file extension project wide. See ["Selective resource copying" on page 6-25](#) for more information on

resources. See [“Setting the output path” on page 5-8](#) for information on output paths for a project.



The automatic source packages feature is on by default and is located on the General page of the Project Properties dialog box. To change the settings, choose Project | Project Properties and choose the General page. Then, check or uncheck the Enable Source Package Discovery And Compilation option. You can also control the level of packages displayed in the project pane by changing the value in the Deepest Package Exposed field. For more information on this feature, press the Help button on the General page (Project | Project Properties).

To minimize the number of package nodes listed, JBuilder automatically displays a subset of the packages in your project. Even though some source packages may not be listed at the top level of your project, JBuilder still builds them.

**Important** After adding new source files to the project, select the refresh button on the project toolbar to update the automatic source packages list.

A <Project Source> node also displays at the top of the project pane when the automatic source packages feature is enabled. This node contains all the source packages and source files in the project, except packages and files that you’ve added manually. You can use this node to quickly filter source packages. See [“Filtering packages” on page 6-23](#) for more on filtering.

**Note** If you have file types in your project that JBuilder doesn’t recognize, you can add them as generic resource files. For more information, see [“Adding unrecognized file types as generic resource files” on page 6-27](#).

For more information on Java packages, see “Chapter 7: Packages” in the Java Language Specification at [http://java.sun.com/docs/books/jls/second\\_edition/html/packages.doc.html#60384](http://java.sun.com/docs/books/jls/second_edition/html/packages.doc.html#60384).

### See also

- [“How JBuilder constructs paths” on page 4-9](#)
- General page of Project Properties dialog box

## Filtering packages

---

This is a feature of  
JBuilder SE and  
Enterprise

JBuilder provides a filtering feature that allows you to exclude packages from the build process. However, if JBuilder’s dependency checker determines that there is a dependency on classes in the excluded packages, those classes are compiled. When you exclude packages from a project, a Package Filters folder displays in the project pane. This folder contains an overview of any filtering applied to the packages in the project. Icons in this folder indicate the filtering applied. See [Table 6.3, “Package filtering icons,” on page 6-25](#) for definitions.

### Important

The automatic source packages feature, which is on by default, must be enabled on the General page of Project Properties. If you have deeply nested packages and are excluding only a few packages, it’s recommended that you increase the number in the Deepest Package Exposed field so more packages are displayed in the project pane. For more information on automatic source packages, see [“Automatic source packages” on page 6-21](#).

A <Project Source> node also displays at the top of the project pane when the automatic source packages feature is enabled. This node contains all the source packages and source files in the project, except packages and files that you’ve added manually. You can use this node to quickly filter source packages. Once this node is excluded, it displays in the Package Filters folder and not at the top of the project pane.

Manually added packages and files can’t be excluded with the Apply Filter command. You must remove them from the project to exclude them from the build process. Also, any buildable nodes that are children of the project node, such as Java source files added by wizards, are compiled and can’t be excluded unless you remove them from the project. Essentially, any files or packages that display above the Package Filters folder are not filtered and are included in the build process.

## Excluding packages

---

To exclude packages from the build process,

- 1 Select the package(s) in the project pane that you want to exclude.
- 2 Right-click and choose Apply Filter or choose Project | Apply Filter.
- 3 Choose one of these menu commands from the submenu:
  - Exclude Package And Subpackages: excludes selected package(s) and their subpackages.
  - Exclude Package: excludes selected package(s) but not their subpackages.

**Note** If the package and subpackages are excluded, the package isn't displayed at the top of the project pane. It's only displayed in the Package Filters folder. If a package is excluded, but a subpackage is included; it's displayed with the appropriate icon at the top of the project pane and in the Package Filters folder.

If you want to exclude all the source packages in the project,

- 1 Right-click the <Project Source> node in the project pane and choose Apply Filter.
- 2 Choose Exclude Package And Subpackages from the submenu.

In some cases, filtering may be a two-step process. For example, if you want to exclude all the packages except for a few subpackages, you would do the following:

- 1 Select the <Project Source> node in the project pane and choose Project | Apply Filter.
- 2 Choose Exclude Package And Subpackages from the submenu.
- 3 Open the Package Filters folder.
- 4 Drill down and select the subpackage(s) you want to include.
- 5 Right-click and choose Apply Filter | Include Package.

Filter settings are saved locally in the project file. Any new filter settings that are applied to a package override the previous filter settings for that package.





## Including packages

---

Once you've excluded packages, there are several ways to include them in the build process again.

- Choose Project | Remove All Filters.
- Right-click the Package Filters folder and choose Remove All Filters.
- Expand the Package Filters folder. Right-click a package or packages, and choose Apply Filter. Then choose one of these menu commands from the submenu:
  - Include Package And Subpackages: includes selected package(s) and their subpackages.
  - Include Package: includes selected package(s) but not their subpackages.

**Table 6.3** Package filtering icons

Icon	Menu command	Description
	Exclude Package And Subpackages	Excludes any selected packages and all their subpackages from the build process.
	Include Package And Subpackages	Includes any selected packages and all their subpackages in the build process.
	Exclude Package	Excludes only the selected packages from the build process but doesn't exclude their subpackages.
	Include Package	Includes only the selected packages in the build process but doesn't include their subpackages.

## Selective resource copying

---

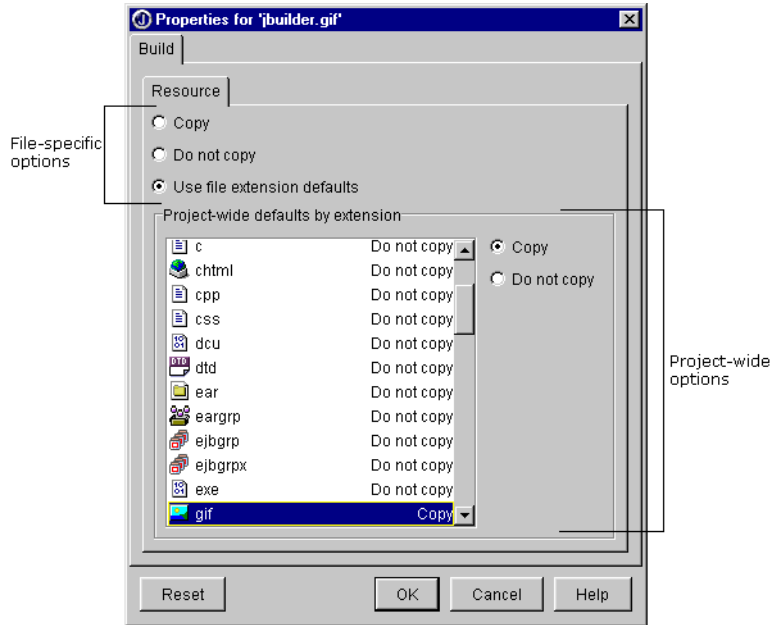
This is a feature of  
JBuilder SE and  
Enterprise

JBuilder copies all known resource types from the project's source paths to the output path during the compile process. By default, JBuilder recognizes all images, sound, and properties files as resources and copies them to the output path. You can override these default resource definitions on individual files or by file extension project wide. See [“Setting the output path” on page 5-8](#) for more information on the output path.

## Individual resource properties

---

To change the default for individual files in a project, select and right-click the file(s) in the project pane, choose Properties, and choose the Resource tab on the Build page.

**Figure 6.1** Properties Resource page

### File-specific options

The top three radio buttons are file-specific options which control the currently selected file(s). These options are:

- **Copy:** copies selected file(s) to the output path.
- **Do Not Copy:** does not copy the selected file(s) to the output path.
- **Use File Extension Defaults:** uses the project-wide defaults as displayed in the Project-wide Defaults By Extension list.

The Copy and Do Not Copy options select an absolute behavior: always copy to the output path or never copy to the output path when the project is built, regardless of whether or not the file type is normally considered a resource.

The third option, Use File Extension Defaults, allows JBuilder to deploy the file based on its file extension in the file list below. This is the default behavior for all newly created files and files in existing projects. The correct extensions for the selected files are automatically selected in the list to highlight the default behavior.

**Important** If the selected files or extensions do not all share the same setting, **none** of the radio buttons in the corresponding group are selected. Selecting one of the radio buttons changes everything to the same value, while leaving none selected allows the differing values to be left alone.

If you change the defaults for individual files and you want to return them to the project-wide defaults, select the files again and choose Use File Extension Defaults.

## Project-wide options

Below the three file-specific options is a list of project-wide defaults by extension and their default deployment behavior. These defaults can be changed on a project-by-project basis. Select one or more extensions and use the radio buttons on the right to change the default behavior for these extensions in the current project. The project-specific options include:

- Copy: copies selected file(s) to the output path.
- Do Not Copy: does not copy the selected file(s) to the output path.

Use the Reset button to return all files in the file extension list to the state they were in when the dialog box was initially displayed. Remember, this does not change your individual file settings to the default.

You can also change these defaults for all future projects in the Default Project Properties dialog box (Project | Default Project Properties).

## Adding unrecognized file types as generic resource files

If you have file types that aren't recognized by JBuilder, you can associate them with the generic resource file type. Then JBuilder will recognize them and include them in the automatic source discovery process. You'll also be able to bundle them into archives. For example, JBuilder doesn't recognize Flash files, but you might want to access them in your project and deploy them to an archive with the rest of your project. To bundle an unrecognized file type in an archive is a two-step process. First, you need to add it as a recognized file type. Then you need to set it to copy to the output path when the archive is built.

- 1 To add unrecognized file types to the list of recognized file types for JBuilder, complete these steps:
  - a Choose Tools | IDE Options and choose the File Types tab.
  - b Choose Generic Resource File in the Recognized File Types list.
  - c Click the Add button, enter the new file extension, and click OK.
  - d Click OK to close the IDE Options dialog box.

### Important

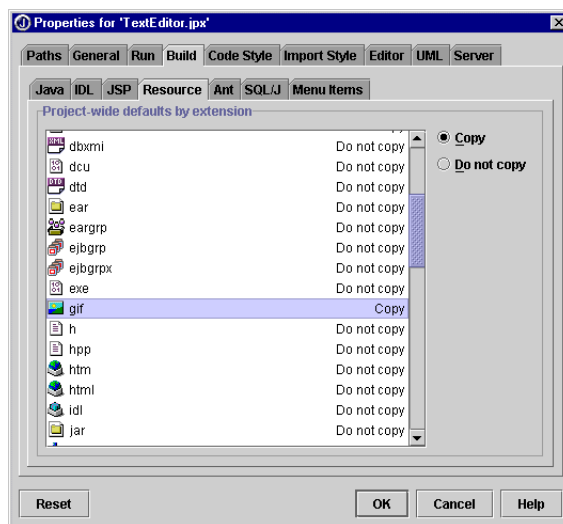
By default, any file type added as a Generic Resource File is **not** included in archives. You need to set the new file type to copy to the output path on the Resource tab of the Build page in Project Properties. Continue on to the next step to do this.

- 2 To bundle the new file type with your archive, complete these steps:
  - a Choose Project | Project Properties and click the Build tab. To bundle the new file type for all future projects, choose Project | Default Project Properties.
  - b Choose the Resource tab and select the new file type in the Recognized File Types list. Notice that by default it's set to Do Not Copy.
  - c Choose the Copy radio button to set this file type to copy to the output path.
  - d Click OK to close the Project Properties dialog box.

## Project Properties Resource page

This is a feature of  
JBuilder SE and  
Enterprise

The Project Properties dialog box has a corresponding Resource tab on the Build page that provides control over the default behaviors for file extensions for the entire project rather than on an individual file basis. Use the Reset button to return all files in the file extension list to the state they were in when the dialog box was displayed.





## Running Java programs

When you're ready to test your program, you can simply run it, or you can run it and debug it at the same time. When you run your program, JBuilder uses the class path to locate all classes your program uses. To understand how JBuilder locates files to run the program, see ["How JBuilder constructs paths" on page 4-9](#) and ["Where are my files?" on page 4-12](#).

There are several ways to run your files. You can run an individual file, such as an applet HTML file or an application file, by right-clicking the file in the project pane and choosing the Run command. You can also run a project by selecting the Run Project button on the main toolbar.

The Run Project button can be configured with preset runtime parameters and saved as drop-down menu selections. See ["Setting runtime configurations" on page 7-6](#). You must have at least one runtime configuration created to be able to run a project using *F9* or the Run toolbar button. When you have multiple configurations, you can choose one of them as the default to use at runtime.

### Running program files

---

To run a program file within a project,

- 1 Save the .java file.
- 2 Choose the .java file containing the `main()` method and do one of the following:
  - Right-click the file in the project pane and select Run. If you have more than one runtime configuration, you'll need to also select the one you want to use from the sub-menu.

- Double-click the file in the project pane to open it, or if opened already, select it to make it the active file. Right-click the file's Name tab in the content pane, and choose Run.
- Click the arrow beside the Run icon on the toolbar and choose a configuration from the drop-down list.
- Select Run | Run Project (*F9*) on the main menu. This runs the main class specified in the default configuration on the Run page of the Project Properties dialog box. See [“Running projects” on page 7-3](#). If no default is selected, and you have just one configuration, it runs the main class specified in that configuration.

In JBuilder SE or Enterprise,

- If the project has more than one runtime configuration, but no default configuration is set, you are prompted to select a configuration before the run operation continues.
- If no configurations exist for the project, the Run page of the Project Properties dialog box will appear so you can create a runtime configuration when you click the Run button or press *F9*. You can run without runtime configurations by right-clicking the runnable file in the Project pane and choosing Run Using Defaults.

**Note** If there is no runtime configuration, after you create one, you must rerun the application.

The program compiles and runs if there are no errors. The build progress is displayed in the status bar and the message pane displays any compiler errors. Before compiling, error messages are displayed in the **Errors** folder of the structure pane. If the program compiles successfully, the Java command line is displayed in the message pane and the program runs.

## Running web files

---

Applets are a feature of all JBuilder editions.

This is a feature of JBuilder SE and Enterprise.

JBuilder also supports running web files, such as JSPs, servlets, SHTML, and HTML, through a web server for a live view of your web application. To run an applet, right-click the HTML file containing the `<applet>` tag and choose Run. To run JSP, SHTML, and HTML files, right-click the file in the project pane and select Web Run.

For servlets, you must first set the Enable Web Run/Debug/Optimize From Context Menu option. If you create your servlets with JBuilder's Servlet wizard, this option is set automatically. To enable this option, right-click the servlet file in the project pane and select Properties. Check the Enable Web Run/Debug/Optimize From Context Menu option on the Web Run page. Then right-click the servlet file and select Web Run.

**See also**

- “Working with applets” in the *Web Application Developer’s Guide*
- “Working with web applications in JBuilder” in the *Web Application Developer’s Guide*
- “Creating servlets in JBuilder” in the *Web Application Developer’s Guide*
- “JavaServer Pages (JSP)” in the *Web Application Developer’s Guide*

## Running projects

---



You can run your project by selecting Run | Run Project, pressing the *F9* shortcut key, or by clicking the Run Project button on the main toolbar. This runs the main class selected in the default runtime configuration. If you have no default runtime configuration set, and have just one configuration, it uses that configuration by default. If you have multiple configurations with no default selected, JBuilder prompts you to choose a runtime configuration.

Runtime configurations are set on the Run page of the Project Properties dialog box, by choosing Run | Configurations from the main menu, or by selecting Configurations from the drop-down menu on the toolbar Run icon. You can also create a runtime configuration as the last step of some wizards, such as the Application, Applet, Servlet, and Test Case wizards.

If a main class has not yet been selected, the dialog box appears for you to make the selection. If you created your file with the Application wizard, the main class is automatically selected.

With JBuilder SE and Enterprise, you can also set run and debug configurations for the Run Project and Debug Project buttons. These configurations are added to the drop-down menus that are accessible from the Down arrows to the right of each buttons, and from the Run menu items.

**See also**

- [“Setting runtime configurations” on page 7-6](#)
- [“Running tests” on page 13-13](#)

## Using the Run command

---

When you choose the Run command (Run | Run Project), or click the Run Project button on the main toolbar, JBuilder runs the program according to the runtime configuration settings in the default or selected runtime configuration. Many of JBuilder’s wizards can create a runtime

configuration and set the main class for you automatically if you choose to do so. The Run command and the Run Project button also execute whatever Build Target is specified in the runtime configuration being used.

To run your project without debugging,

- 1 Save your files.
- 2 Make sure a main class is selected in the runtime configuration you're going to use. You can add or edit runtime configurations from the Run page of the Project Properties dialog box if none are already set.
- 3 Choose Run | Run Project, press *F9*, or click the Run button on the toolbar.

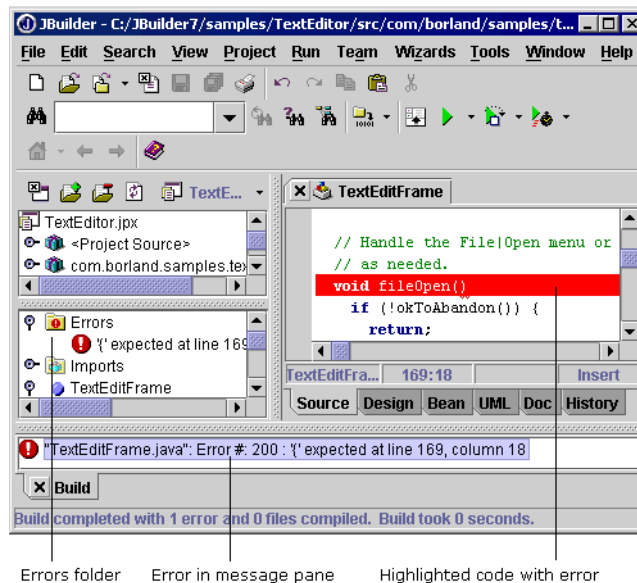


**Tip**

Alternatively, you can right-click the project in the project pane and choose Clean, Make, or Rebuild first, then click the Run button on the toolbar.

JBuilder executes the Build Target you chose and runs your program. Any errors during compile are displayed in the message pane at the bottom of the AppBrowser. If there are errors, compiling stops so you can fix the errors and try again. Select an error in the message pane or in the **Errors** folder in the structure pane to highlight the code in the source pane. For help on an error message, select the error in the message pane and press *F1*.

**Figure 7.1** Error messages in the AppBrowser



**See also**

- [Chapter 6, “Building Java programs”](#)
- [Chapter 5, “Compiling Java programs”](#)

## Running grouped projects

---

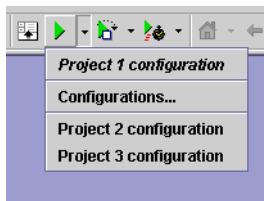
To run projects that are in a project group (JBuilder Enterprise only), right-click the project in the project pane and choose Run. The Run button on the toolbar only runs the active project, not the project group. The Run Configuration drop-down list on the toolbar displays the run configurations for the active project, and any additional runtime configurations you have specified be available to the project group.

You can specify which project group runtime configurations will be added to the Run Configuration drop-down list drop-down list when you create or edit each runtime configurations for the projects in the group.

To surface the runtime configurations for the project,

- 1 Open the project group.
- 2 Right-click a project in the group and choose Properties.
- 3 Click the Run tab in the Project Properties dialog box, then select the desired runtime configuration. (See [“Setting runtime configurations” on page 7-6](#) for information on creating and editing runtime configurations.)
- 4 Make sure the Group box is checked for that configuration.(Group is check by default.)
- 5 Repeat this for any other configurations in this project, or other projects in the group you want available on the Run Configurations drop-down list, then click OK.

Now, when you click the down arrow beside the Run icon on the toolbar, in addition to the runtime configurations for the current project, you will see all the exposed runtime configurations from the other projects in the group.



For more information on creating and using project groups, see [“Chapter 3, “Working with project groups.”](#)

## Running OpenTools

---

JBuilder has an OpenTool runtime configuration type in JBuilder Enterprise that lets you run, debug, and optimize your OpenTool project in JBuilder just like other projects. You no longer have to exit JBuilder,

create a JAR file and copy it to the <JBuilder home>\lib\ext or <JBuilder home>\patch directory, then restart JBuilder.

When you run an OpenTool project using the OpenTool runner, a temporary config file is generated in the <outpath>\..\config-temp directory, and a new instance of JBuilder is started using this temporary config file. This configuration file will be removed when the tracker is closed.

To use this OpenTool runner, you must create a new configuration for your OpenTool project that is of the OpenTool type by doing the following:

- 1 Open your OpenTool project in JBuilder.
- 2 Choose Run | Configurations, or click the down arrow on the Run button and choose Configurations to open the Run page of the Project Properties dialog box.
- 3 Click New to open the Runtime Properties dialog box.
- 4 Type a name for the configuration, and choose a Build Target.
- 5 Specify the location of the files to use at runtime: the ones in your OpenTool project directory, or those in a JAR file.
- 6 Specify any parameters you want passed to VM or JBuilder at runtime, as well as the location of your <JBuilder Home> path.
- 7 Check Override Existing Classes and Resources if you want the OpenTool classes and resources to be first on the classpath.
- 8 Click OK to return to the Run page of the Project Properties dialog box.
- 9 Check any of the boxes that apply for this configuration (Default, Context Menu, or Group), and use the Move Up or Move Down buttons to move it to the preferred location in the list. This is the same order the configurations will be listed wherever the runtime configurations are listed in the JBuilder menus and drop-down lists.
- 10 Click Ok when you're finished.

For additional help when setting these runtime configurations, click the Help button at the bottom of the Runtime Properties dialog box.

## Setting runtime configurations

---

JBuilder Personal can have one runtime configuration only.

Runtime configurations (Run | Configurations) are preset runtime parameters. Using preset parameters saves you time when running and debugging, because you only have to set the parameters once. With preset configurations, each time you run or debug you simply select the desired configuration.

You manage runtime configurations on the Run Page of the Project Properties dialog box. From there, you can add, edit, copy, or delete configurations, and control the order the configurations display in the selection menus. You can also designate one of the configurations as the default configuration to use when you choose run or debug without selecting a configuration, and you can specify which configurations will appear on the context-menus for a particular project, or for a project group.

To set runtime configurations, open the Run page of the Project Properties dialog box one of the following ways:

- Choose Project | Project Properties, and click the Run tab.
- Choose Run | Configurations.
- Click the down arrow beside the Run or Debug buttons and choose Configurations from the drop-down menu.
- Right-click the project file in the project pane and choose Properties, then click the Run tab.

Some of the JBuilder wizards, such as the Application, Applet, Servlet, or Test Case wizard, give you the option of creating a runtime configuration.

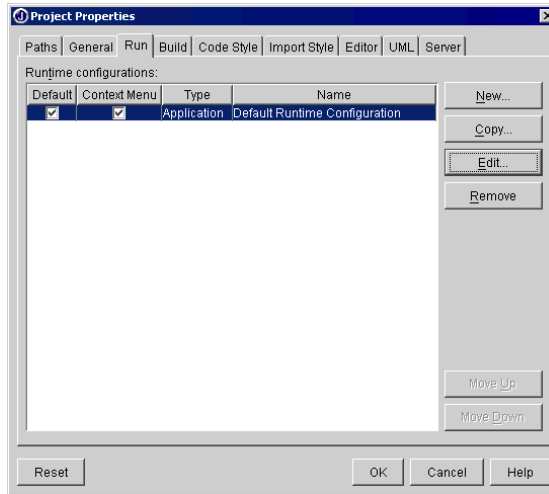
- If you have only one runtime configuration, JBuilder uses that configuration when you choose Run | Run Project, (*F9*), or Run | Debug Project (*Shift-F9*).
- If you specify a default configuration, JBuilder displays that configuration in bold (or italics, depending on your look and feel) in the Run and Debug toolbar button drop-down lists, and uses that configuration when you run or debug your project.
- If you do not specify a default configuration, when you choose Run or Debug you are prompted to select which configuration to use.
- If you have no existing configurations for the type of file being run, when you choose Run from the menu, press *F9*, or click the Run button on the toolbar, the Project Properties Run page displays so you can create a configuration at that time. However, if you right-click the runnable file in the project pane and choose Run, Builder will automatically run it based defaults it assumes that are appropriate for that type of file.

## Creating a runtime configuration

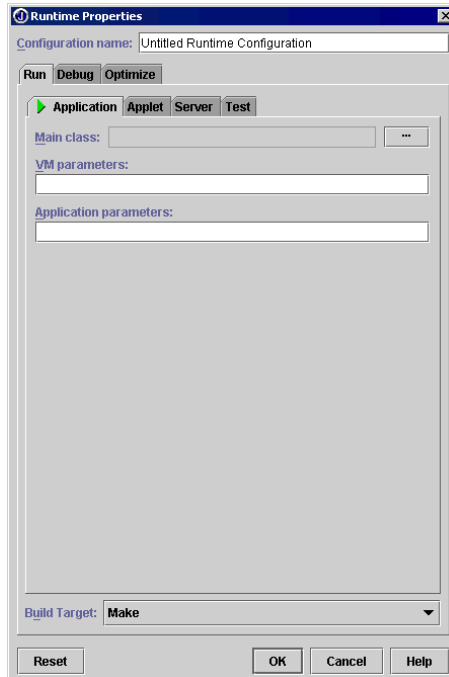
---

To create a runtime configuration,

- 1 Choose Run | Configurations, or choose Project | Project Properties and click the Run tab.



- 2 Click New to open the Runtime Properties dialog box.



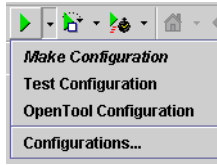


- 3 Type in a name for the configuration if you want something other than the default. This name will be added to the Run Project and Debug Project configuration drop-down lists and context menus.
- 4 Select a Build Target for this configuration from the drop-down list of build options.
- 5 Select the appropriate runtime configuration type for your project, such as Application, Applet, Server, Test, OpenTool, or MIDlet if you have MobileSet installed. (The types of runtime configurations are mutually exclusive from each other.) Specify the main class to run, if applicable, and enter any desired VM and Application Parameters. For more information on the available options for a specific runner type, select that type, then click the Help button at the bottom of the dialog box.

For a description of the configuration types, see [“Runtime configuration types” on page 7-12](#).

- 6 Click the Debug tab and choose any desired debug options.
- 7 Click the Optimize tab if you have Optimizeit installed and set those options.
- 8 Click OK to return to the Run page of the Project Properties dialog box and make any changes to the checkboxes on this page.
  - Check the Default box for the runtime configuration to use when you click the Run or Debug button, or press (F9) or (Shift-F9).
  - Check the Context Menu box for each configuration you want displayed on the project’s context menu.
  - Check the Group button if this project belongs to a group, and you want the configuration available on the configuration list for the project group.
- 9 Add any additional configurations you would like. When you have all the configurations set up, put them in the desired order using the Move Up and Move Down buttons. The way they display in this list is the way they appear on the Run/Debug drop-down menus.
- 10 Check which configuration you want as the default configuration in the Default column. If you have multiple configurations and do not select one as the default, JBuilder will prompt you to select one when you run the application.
- 11 Place a check mark in the Context Menu column beside each configuration you want to appear on the configuration context menus.

To choose a configuration at runtime, click the Down arrow next to the Run or Debug icons on the main toolbar.



If you get errors, runtime exceptions, or other program misbehavior, you may want to debug your program as you run it to find the problems. You do that by running the program in JBuilder's integrated debugger.

### See also

- [Chapter 8, "Debugging Java programs"](#)

## Editing a runtime configuration

---

You can modify everything about an existing runtime configuration except the type. If you want a different type of configuration, you must create a new one.

To edit an existing runtime configuration,

- 1 Choose Run | Configurations, or click the down-arrow next to the Run button on the toolbar and choose Configurations.
- 2 Select an existing runtime configuration on the Run page and click Edit, or double-click the configuration name to open the Runtime Configurations Properties dialog box.
- 3 Make the desired changes to the configuration in the Runtime Configurations Properties dialog box, then click OK to exit it.
- 4 Make any additional changes on the Run page to the runtime configurations, such as moving them in the list to change the order they appear on the drop-down list, or changing which configuration is the default configuration. (Click the Help button at the bottom of the Run page for more details on these options.)
- 5 Click OK when you are finished.

## Build Targets

---

For each runtime configuration you create, you can specify which Build Target to execute prior to running in the Runtime Properties dialog box. JBuilder provides a set of standard targets from which to choose, and additional ones are added to the list depending on your project. Below are the items typically found on the Build Target list:

- **<None>**  
Runs without compiling first.

- **Make**

Make establishes dependencies among the stand-alone phases in this order: Pre-Compile, Compile, Post-Compile, Package, and Deploy.

- **Rebuild**

Rebuild has Clean and Make as dependencies.

- **Clean**

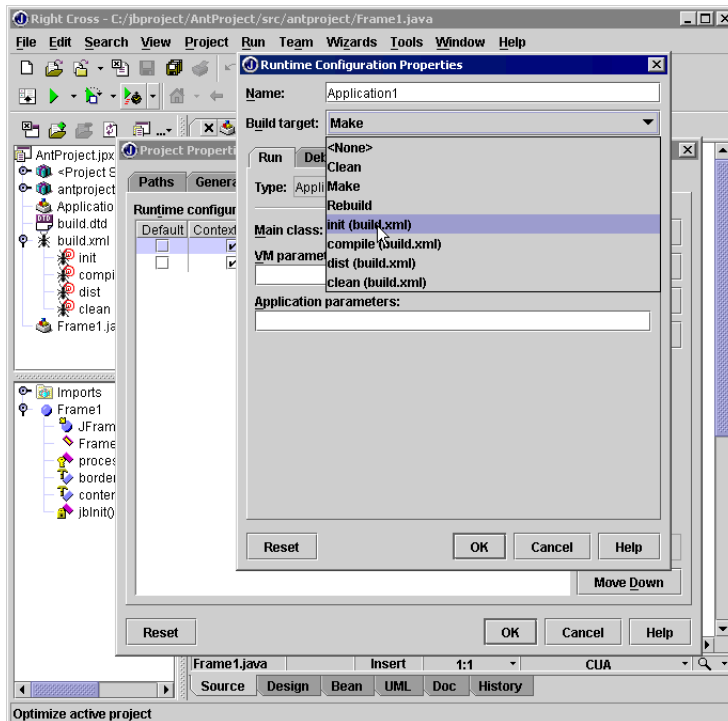
Clean removes all build output of the other targets, such as the `classes` directory, JARs, WARs, and so on. What Clean removes is dependent upon the node selected.

- **External Build Task** (JBuilder Enterprise only)

If you have created an external build task, it will display in the list of available targets from which you can choose for the runtime configuration. For information, see [“Creating external build tasks” on page 6-16](#).

- **Ant file targets** (JBuilder Enterprise only)

If you have any Ant build files in your project, the targets inside the Ant file are added to the list of available targets in the Runtime Configurations dialog box.



Also, if you have extended your build system through OpenTools, any of those build task targets will appear in the list.

### See also

- [Chapter 6, “Building Java programs”](#)
- [Chapter 5, “Compiling Java programs”](#)

## Runtime configuration types

---

The available runtime configuration types vary by JBuilder edition. Each type presents a different set of runtime properties in this dialog box.

- **Application**

Select the Application type for an application and click the ellipsis button. Browse to the class file containing the `main()` method. The main class must be in the current project.

- **Applet**

Select the Applet type for an applet project and do one of the following:

- Select Main Class and click the ellipsis button to browse to the class containing the `init()` method. This option runs the applet in JBuilder’s applet viewer, `AppletTestbed`.
- Select HTML File and click the ellipsis button to browse to the applet HTML file containing the `<applet>` tag. The HTML option runs the applet in Sun’s **appletviewer**.

- **Server**

Select the Server type and set the parameters for the server being used. See “Working with web applications in JBuilder” in the *Web Application Developer’s Guide*. Click the Help button for more information on setting the server runtime parameters.

- **Test**

Select the Test type if you have a runtime configuration for the Test Case, or Test Suite. Choose the main class for the test suite class, any VM parameters, and the desired test runner to use. Click the Help button for more information on setting the test runtime parameters.

- **OpenTool**

Select the OpenTool type to set up a runtime configuration for running, debugging, and optimizing an OpenTool project directly from within JBuilder.

The OpenTool runtime configuration type lets you run, debug, and optimize your OpenTool project in JBuilder just like other projects. You

no longer have to exit JBuilder, create a JAR file and copy it to the <JBuilder home>\lib\ext or <JBuilder home>\patch directory, then restart JBuilder.

When you run an OpenTool project using the OpenTool runner, a temporary config file is generated in the <outpath>\..\config-temp directory, and a new instance of JBuilder is started using this temporary config file. This configuration file will be removed when the tracker is closed.

- **MIDlet**

Select the MIDlet type (if you have MobileSet installed) to set up a runtime configuration for running and debugging a J2ME MIDlet in JBuilder.

## Running programs from the command line

---

Running your program outside JBuilder requires that you put all the libraries required by your program on your CLASSPATH or add them to the **-classpath** argument to the `java` command. For example,

```
java -classpath /<jbuilder>/lib/dbswing.jar
/<home>/jbproject/mypackage/classes/mypackage.application1
```

In this example,

- <jbuilder> = the name of the JBuilder directory
- <home> = your user home directory, for example, c:\winnt\profiles\  
<username> directory

## Running a deployed program from the command line

---

After deploying the program using the Archive Builder or the **jar** tool, you can run the JAR file from the command line.

- 1 Open the command-line window.

**Tip** For Windows, use backslashes (\) in all commands discussed here.

- 2 Enter the command in the following form on one line at the prompt from any location:

```
java -classpath classpath package-name.main-class-name
```

**Note** The <jdk>/bin/ directory must be on your path. <jdk> represents the name of the JDK home directory.

For example, the command could look something like this:

```
java -classpath /<home>/jbproject/hello/classes/HelloWorld.jar
hello.HelloWorldClass
```

In this example, `<home>` represents your home directory, such as `c:\winnt\profiles\<username>`.

You can also use the **-jar** option:

```
java -jar HelloWorld.jar
```

**Note** You must first change to the directory that contains the `.jar` file before running this command with the **-jar** option.

### See also

- [Chapter 6, “Building Java programs”](#)
- [Chapter 5, “Compiling Java programs”](#)
- [Chapter 15, “Deploying Java programs”](#)
- The JAR tutorial at <http://java.sun.com/docs/books/tutorial/jar/index.html>
- [Appendix B, “Using the command-line tools”](#)

# Debugging Java programs

Debugging is the process of locating and fixing errors in your program. JBuilder's integrated debugger lets you debug within the JBuilder environment. Many debugger features are accessed through the Run menu. You can also use context menus, both in the editor and in the debugger, to access debugger features.

When the debugger pauses your program, you can look at the current values of the program data items. Modifying data values during a debugging session provides a way to test hypothetical bug fixes during a program run. If you find that a modification fixes a program error, you can exit the debugging session, fix your program code accordingly, and recompile to make the fix take effect. If you are debugging with JDK 1.4 or higher, you do not need to exit the debugging session to have the fix take effect. See [“Modifying code while debugging” on page 8-64](#) for more information.

For a tutorial on debugging, see [Chapter 17, “Tutorial: Compiling, running, and debugging.”](#) If you have JBuilder Personal, see *Building Applications with JBuilder* in online help for a version of the tutorial that is specifically designed for JBuilder Personal features.

Additional information and tips are available for these specific debugging topics:

- If your program uses a `JDataStore` component, see the “Troubleshooting” chapter of the *JDataStore Developer's Guide*. (JBuilder Enterprise)
- If you are debugging a distributed application, see [Chapter 9, “Remote debugging.”](#) (JBuilder Enterprise)
- If you are debugging a unit test, see [Chapter 13, “Unit testing.”](#) (JBuilder Enterprise)

To debug outside JBuilder, use the **jdb** tool in the `<jdk>/bin/` directory. See the JDK documentation at <http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html> for more information on this tool.

## Types of errors

---

The debugger can help find runtime errors and logic errors. If you find or suspect a program runtime or logic error, you can begin a debugging session by running your program under the control of the debugger.

### Runtime errors

---

If your program contains valid statements, but the statements cause errors when they're executed, you've encountered a runtime error. For example, your program might be trying to open a nonexistent file, or it might be trying to divide a number by zero.

Without a debugger, runtime errors can be difficult to locate, because the compiler doesn't tell you anything about them. Often, the only clues you have to work with are the results of the run, such as the screen appearance, and the error message generated by the runtime error.

Although you can find runtime errors by searching through your program source code, the debugger can help you quickly track down these types of errors. Using the debugger, you can run to a specific program location. From there, you can begin executing your program one statement at a time, watching the behavior of your program with each step. When you execute the statement that causes your program to fail, you have pinpointed the error. From there, you can fix the source code and resume testing your program.

If your program throws a runtime exception, it will print a stack trace in the Console output, input, and errors view. You can click the underlined file name and line number in the stack trace to go to the line of code in the source file listed in the trace. (This is a feature of JBuilder SE and Enterprise.)

### Logic errors

---

Logic errors are errors in design and implementation of your program. Your program statements are valid (they do something), but the actions they perform are not the actions you had in mind when you wrote the code. For example, logic errors can occur when variables contain incorrect



values, when graphic images don't look right, or when the output of your program is incorrect.

Logic errors are often the most difficult type of errors to find because they can show up in places you might not expect. To be sure your program works as designed, you need to thoroughly test all of its aspects. Only by scrutinizing each portion of the user interface and output of your program can you be sure that its behavior corresponds to its design. As with runtime errors, the debugger helps you locate logic errors by letting you monitor the values of your program variables and data objects as your program executes.

## Overview of the debugging process

---

After program design, program development consists of cycles of program coding and debugging. Only after you thoroughly test your program should you distribute it. To ensure that you test all aspects of your program, it's best to have a thorough test and debug plan.

One good debugging method involves breaking your program down into different sections that you can systematically debug. By closely monitoring the statements in each program section, you can verify that each area is performing as designed. If you find a programming error, you can correct the problem in your source code, recompile the program, and then resume testing.

JBuilder Enterprise allows you to debug non-Java source, including JavaServer Pages (JSPs). You can set a breakpoint in a non-Java source file and debug that file either locally or remotely. You can also switch between viewing the non-Java source or the generated Java code. For more information, see [“Debugging non-Java source” on page 8-27](#).

**Note** You can debug with any JDK that supports the JPDA debugging API. Usually, you will debug with the version of the JDK that JBuilder ships with. (This is the default JDK selected for the project, if no other ones have been defined and selected.)

## Creating a runtime configuration

---

Before running or debugging, you need to create a runtime configuration. A runtime configuration is a set of pre-configured parameters. Using preset parameters saves you time when running and debugging, because you only have to set the parameters once. With preset configurations, each time you run or debug you simply select the desired configuration. You

set debugger options, such as Smart Step settings and remote debugging options, through a runtime configuration.

To create and manage configurations, you use the Runtime Configuration Properties dialog box. For more information on runtime configurations, see [“Setting runtime configurations” on page 7-6](#). For more information on the debugger options, see [“Setting debug configuration options” on page 8-68](#). (Multiple runtime configurations are a feature of JBuilder SE and Enterprise.)

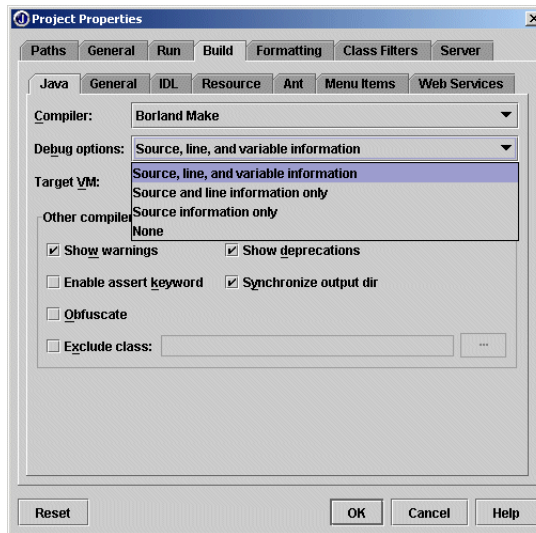
## Compiling the project with symbolic debug information

Stripping debug information is a feature of JBuilder SE and Enterprise

Before you can begin a debugging session, you need to compile your project with symbolic debug information. Symbolic debug information enables the debugger to make connections between your program’s source code and the Java bytecode that is generated by the compiler.

By default, JBuilder includes symbolic debug information when you compile. To be sure that this option is set for the current project,

- 1 Select Project | Project Properties to open the Project Properties dialog box.
- 2 Choose the Build tab, then the Java tab. The Build | Java page looks like this (the Debug Options are displayed):



**3** Select one of the following options in the Debug Options drop-down list:

- **Source, Line, And Variable Information:** includes debugging information with source name, line number, and local variable information in the `.class` file when you compile, make, or rebuild a node.
- **Source And Line Information Only:** includes debugging information with source name and line number only in the `.class` file when you compile, make, or rebuild a node.
- **Source Information Only:** includes debugging information with source name only in the `.class` file when you compile, make, or rebuild a node.
- **None:** No debugging information is included. You can still debug with this option—the `this` object is still available for debugging. By selecting this option, you can significantly reduce the class to the smallest possible size.

**Tip** To set this option for all new projects, choose Project | Default Project Properties and select one of the first three debug options on the Build page. (Setting the default project properties does not affect existing projects.)

**Note** You won't be able to view variable information in the Java API classes because they were compiled with source and line information only. You can, however, trace into these classes. To learn how to trace into classes, see [“Controlling which classes to trace into” on page 8-37](#).


When you generate symbolic debug information, the compiler stores this information in the associated `.class` file. This can make the `.class` file substantially larger than compiling without debugging information. You may want to turn this option off before deployment.

To make compiling before debugging automatic, set the Build Target at the top of the Runtime Configuration Properties dialog box (when you set the debugging options for your runtime configuration) to Make. This option automatically compiles your project before running it under the debugger's control. If this option is set to `<None>`, JBuilder will not compile your program before debugging, so that your source files and class files can be out of sync. For more information on the build target, see [“Build Targets” on page 7-10](#).

## Starting the debugger

Once you've created a runtime configuration and compiled your project with debug information, you can start the debugger with one of the following menu options. For information on runtime configurations, see [“Setting runtime configurations” on page 7-6](#).

**Table 8.1** Menu commands to start debugger

Command	Shortcut	Description
Run   Debug Project	<i>Shift + F9</i> or 	Starts the program in the debugger using the default or selected configuration. Execution is suspended at a breakpoint or at the first line of code where user input is required, whichever comes first.
Run   Step Over	<i>F8</i>	Suspends execution at the first line of executable code.
Run   Step Into	<i>F7</i>	Suspends execution at the first line of executable code.

### To debug

- A single class file in your project, and not the entire project, choose the source file in the project pane and right-click. Choose the Debug command for the desired configuration.
- A web application, right-click the servlet or JSP file and choose the Web Debug command for the desired configuration. For more information, see “Web debugging your servlet or JSP” in the “Working with web applications” chapter of the *Web Applications Developer's Guide*. (Web application development is a feature of JBuilder Enterprise.)
- A unit test, right-click the test in the project pane and choose Debug Test. For more information on unit testing, see [Chapter 13, “Unit testing.”](#) (Unit testing is a feature of JBuilder Enterprise.)

Each time you start the debugger, you are starting a debugging session. For more information, see [“Debugging sessions” on page 8-9](#).

#### Note



To select a runtime configuration for a debugging session, click the down-facing arrow to the right of the Debug Project button on the main toolbar before you begin. If you don't specifically select a configuration, you use the default configuration defined on the Run page of the Project Properties dialog box. (Multiple runtime configurations are a feature of JBuilder SE and Enterprise.)

## Starting the debugger with the -classic option

In versions of the JVM below 1.3.1, the `-classic` option improved the performance of the debugger. This option does not apply to newer VMs; for example, JDK 1.4x and JDK 1.3.1 on Solaris do not require use of the `-classic` option. In these versions, the VM uses “full-speed debugging,” allowing improved performance.

If you’re using a JVM below 1.3.1 for your project, the Always Debug With -Classic option in the Configure JDKs dialog box (Tools | Configure JDKs) provides improved performance. JBuilder automatically checks to see if this option will improve your performance, then checks or unchecks this box according to what will give you the best results. This feature is available in all editions of JBuilder.

In performing its evaluation, JBuilder performs two checks:

- 1 Do you have the Classic VM?
- 2 If present, is the JVM a version earlier than 1.3.1?

This selection is overridden when you define VM parameters such as `native`, `hotspot`, `green`, or `server`.

## Running under the debugger’s control

---

When you run your program under the control of the debugger, it behaves as it normally would—your program creates windows, accepts user input, calculates values and displays output. The debugger pauses your program, allowing you to use the debugger views to examine the current state of the program. By viewing the values of variables, the methods on the call stack, and the program output, you can ensure that the area of code you’re examining is performing as it was designed to.

As you run your program under the debugger’s control, you can watch the behavior of your application in the windows it creates. Position the windows so you can see both the debugger and your application window as you debug. To keep windows from flickering as the focus alternates between the debugger views and those of your application, arrange the windows so they don’t overlap.

## Pausing program execution

---

When you're in the debugger, your program is in one of two possible states: *running* or *suspended*.



- Your program is running when the Pause button is available on the debugger toolbar.
- Your program is suspended when you click the Pause button. When your program is suspended, you can examine and modify data values. The stepping buttons on the debugger toolbar become available. Hitting a breakpoint or stopping by stepping also pauses execution.



To resume program execution, choose the Resume Program button on the debugger toolbar. When the debugging session is over, this button becomes the Restart Program button and restarts the session.



While your program is suspended, you can modify code and resume execution at any active frame. For more information, see [“Modifying code while debugging” on page 8-64](#).

## Ending a debugging session

---



To end the current debugging session and release the program from memory, choose the Reset Program button.

You can also exit the application to close the debugging session. To remove the debugging session tab, click the X on the tab or right-click the tab and choose Remove.

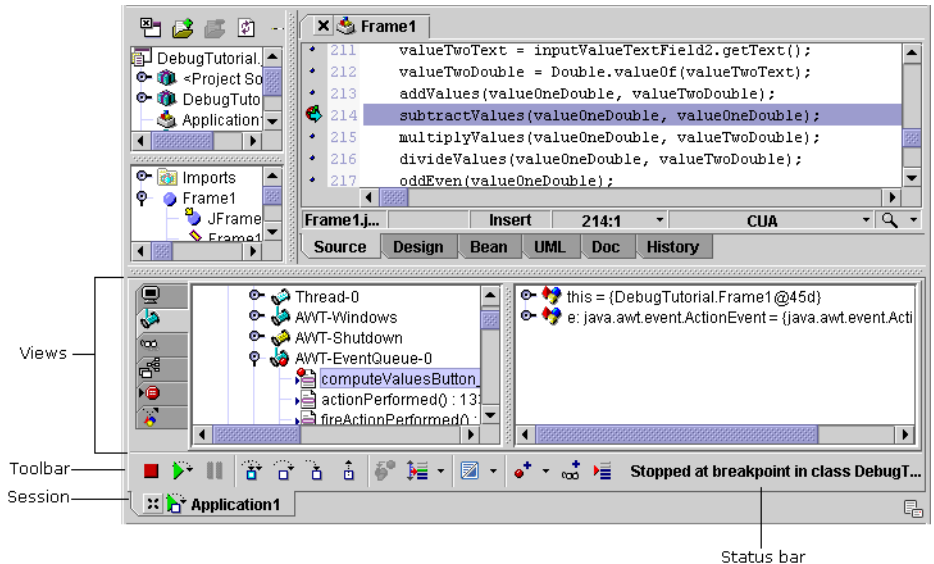
## The debugger user interface

---

If the project or class compiles successfully, the debugger is displayed at the bottom of the AppBrowser.

- Horizontal tabs, along the bottom of the AppBrowser, represent debugging sessions. Each tab represents a new session.
- Vertical tabs, on the lower left of the AppBrowser, represent the debugger views. The views are displayed for the currently selected debugging session. Each view displays icons to indicate the state and type of the item selected in the view.
- The debugger toolbar is displayed for the currently selected debugging session.

Figure 8.1 The debugger user interface




## Debugging sessions

The debugger allows you to debug multiple processes in multiple debugging sessions. Processes can be in the same JBuilder project, or in different ones. This allows for debugging both a client and a server process at the same time, in the same JBuilder instance.

Watches, breakpoints and classes with tracing disabled are stored per individual project. All breakpoints and watches apply to all processes in a project. Breakpoints can be selectively disabled for a runtime configuration.

When you use commands on the Run menu other than Run Project, Debug Project and Configurations, you are continuing in the selected debugging session. When you use buttons on the debugger toolbar, you are also continuing in the selected session.

 To end the current debugging session and release the program from memory, choose the Reset Program button. You can also end the session by clicking X on the debugging session tab or by right-clicking the tab and choosing Remove Tab. Although you will be prompted to stop the process before the tab is removed, it's a good idea to use Run | Reset Program first.

## Debugger views

The debugger views allow you to look inside your program and see what is going on. You use debugger views to examine and change data values, trace backward and forward through your program, examine the internal processing of a method and the call to that method, and follow an individual thread in your program.

Debugger views are displayed along the left side of the debugger UI. To select a view, choose its tab on the left side of the debugger. Views (except the Console output, input, and errors view) can also be displayed as floating windows. Floating windows allow you to see multiple debugger views at the same time, rather than having to switch back and forth between them. (Floating windows are a feature of JBuilder SE and Enterprise.)



- To display a view as a floating window, right-click an empty area of the view, and choose Floating Window.
- To close the floating window, click the Close button in the floating window or right-click an empty area of the view and uncheck Floating Window.
- To restore the default view order after you close a floating window, right-click an empty area of a view and choose Restore Default View Order.

Debugger views also have context menus. Commands on these menus often duplicate those on the Run or View menus, and allow you to easily control the debugger.

Additionally, each debugger view displays a variety of icons to indicate the state of the selected item. For example, a breakpoint can be disabled, verified, unverified, or invalid—each state is indicated visually by a small icon in the left margin of the view.






The debugger views and icons are described below.

**Table 8.2** Debugger views

Tab	View	Description
	Console view	Displays output from the program and errors in the program. Also allows you to enter any input that the program requires. The image displayed on the icon changes if there is any output from the program and if any error messages are displayed.
	Threads, call stacks, and data view	Displays the thread groups in your program. Each thread group expands to show its threads and contains a stack frame trace representing the current call sequence. Each stack frame can expand to show available data elements that are in scope. (Static data is not displayed in this view but is displayed in the Loaded classes and static data view.)



**Table 8.2** Debugger views (continued)

Tab	View	Description
	Synchronization monitors view	Shows synchronization monitors used by the threads and their current state, which is useful in detecting deadlocked situations. This view is only available if the current VM supports it. The HotSpot VM does not support this feature. The ability to detect deadlocked threads is a feature of JBuilder Enterprise.
	Data watches view	Displays the current values of data members that you are tracking.
	Loaded classes and static data view	Displays the classes currently loaded by the program. Expanding a class shows static data, if any, for that class. If a package is displayed in the tree, the number of classes loaded for that package is displayed.
	Data and code breakpoints view	Shows all the breakpoints set in the file and their current state. This view is also available from Run   View Breakpoints before the debugging session begins.
	Classes with tracing disabled view	Displays an alphabetically ordered list of classes and packages not to step into. This view is also available from Run   View Classes With Tracing Disabled before the debugging session begins.

**Tip** You can float all debugger views except the Console view by right-clicking the message pane and selecting Floating Window. (This is a feature of JBuilder SE and Enterprise.)

## Console output, input, and errors view






The Console output, input, and errors view displays output from the program and errors in the program. It also allows you to enter any input that the program requires. When the Console tab is not selected, the icon changes if there is any output from the program or if any error messages are displayed.

Runtime exceptions are displayed in this view. To open the file in which a runtime exception occurred and position the cursor on line number of the exception, click the underlined name of the file. (This is a feature of JBuilder SE and Enterprise.)

In this view, error output is displayed in red font. Standard output is displayed in black font.

**Table 8.3** Icons in Console view

Icon	Description
 (Green)	Output messages have been written to the view.
 (Red)	Error messages have been written to the view.
 (Black)	No output in the view, or the view is in the foreground.

**Table 8.4** Context menu in Console view

Command	Description
Clear All	Clears all messages in the view.
Copy All	Copies the contents of the view to the clipboard.
Copy Selected	Copies the selected content to the clipboard.
Word Wrap	Wraps long lines in the output.

## Classes with tracing disabled view





The Classes with tracing disabled view displays an alphabetically ordered list of classes and packages not to step into. This information is available before you begin debugging from the Run | View Classes With Tracing Disabled command.

By default, when you begin a debugging session, tracing into all classes displayed in the view is disabled. This prevents the debugger from tracing into the libraries that are provided with JBuilder, as well as the JDK classes, allowing you to concentrate on your code, rather than on code that has already been debugged.

For information about controlling what classes are traced into, see [“Controlling which classes to trace into” on page 8-37](#).

**Note** In JBuilder Personal, only three classes (`java.lang.Object`, `java.lang.String` and `java.lang.ClassLoader`) are added to this view. You cannot add, modify or delete items in the list, however, you can choose to step or not step into those classes. See [“Using Smart Step” on page 8-34](#) for more information.

**Table 8.5** Icons in the Classes with tracing disabled view

Icon	Description
 (Gray)	Tracing is disabled for the selected class or package.
 (Colored)	Tracing is enabled for the selected class or package.

**Table 8.6** Context menu with class/package selected in Classes with tracing disabled view

Command	Description
Edit Class/Package	Displays the Edit Disable Tracing Class/Package dialog box, where you can use wildcards to edit the class or package or open the Select Class Or Package dialog box. If you select a package, all classes in that package won't be stepped into. (JBuilder SE and Enterprise)
Step Into Class/Package	Allows the selected class or package to be traced into. If you select a package, all classes in that package will be stepped into.
Remove Class/Package	Removes the selected class or package from the view. This allows the selected class or package to be traced into. (JBuilder SE and Enterprise)

**Table 8.7** Context menu with no selection in Classes with tracing disabled view

Command	Description
Floating Window	Turns the view into a floating window. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Restore Default View Order	Restores the default order of the debugger views. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Add Class Or Package	Displays the Select Class Or Package dialog box, where you choose the class or package to add to the view. This prevents the debugger from tracing into that class or package. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Remove All	Removes all classes and packages from the view. All packages and classes, including those in JBuilder and JDK libraries, will be traced into. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)

## Data and code breakpoints view



The Data and code breakpoints view shows all the breakpoints set in the file and their current state. This information is also available before you begin debugging with the Run | View Breakpoints command.

For more information about breakpoints, see [“Using breakpoints” on page 8-40](#).

**Table 8.8** Icons in Data and code breakpoints view

Icon	Description
(Red)	An unverified breakpoint.
	A verified breakpoint.
	An invalid breakpoint.
	A breakpoint that has been disabled for a configuration. (JBuilder SE and Enterprise)
	A field breakpoint. A field is a Java variable that is defined in a Java object. (JBuilder SE and Enterprise)

**Table 8.9** Context menu with breakpoint selected in Data and code breakpoints view

Command	Description
Go To Breakpoint	Goes to the selected breakpoint in the source code. This is useful if several files are open in the editor and you want to quickly locate the breakpointed line of code. This command is available when a breakpoint is selected.
Enable Breakpoint	Enables the selected breakpoint.

**Table 8.9** Context menu with breakpoint selected in Data and code breakpoints view

Command	Description
Disable For Configuration	Disables the selected breakpoint for the selected configuration. This command is only available if you have configured one or more runtime configurations. By default, every breakpoint is applied to all defined configurations. This command is available when a breakpoint is selected. (JBuilder SE and Enterprise)
Remove Breakpoint	Removes the selected breakpoint.
Breakpoint Properties	Displays the Breakpoint Properties dialog box, where you set properties for the selected breakpoint.
Break On Read	Forces the debugger to stop when the selected field breakpoint is about to be read. A field is a Java variable that is defined in a Java object. This command is available when a field breakpoint is selected. (JBuilder SE and Enterprise)
Break On Write	Forces the debugger to stop when the selected field breakpoint is about to be written to. A field is a Java variable that is defined in a Java object. This command is available when a field breakpoint is selected. (JBuilder SE and Enterprise)

**Table 8.10** Context menu with no selection in Data and code breakpoints view

Command	Description
Floating Window	Turns the view into a floating window. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Restore Default View Order	Restores the default order of the debugger views. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Add Line Breakpoint	Displays the Add Line Breakpoint dialog box, where you add a line breakpoint.
Add Exception Breakpoint	Displays the Add Exception Breakpoint dialog box, where you add an exception breakpoint. (JBuilder SE and Enterprise)
Add Class Breakpoint	Displays the Add Class Breakpoint dialog box, where you add a class breakpoint. (JBuilder SE and Enterprise)
Add Method Breakpoint	Displays the Add Method Breakpoint dialog box, where you add a method breakpoint. (JBuilder SE and Enterprise)
Add Cross-Process Breakpoint	Displays the Add Cross-Breakpoint dialog box, where you add a cross-process breakpoint. (JBuilder Enterprise)
Disable All	Disables all breakpoints.
Enable All	Enables all breakpoints.
Remove All	Removes all breakpoints.

## Threads, call stacks, and data view



The Threads, call stacks, and data view displays the current status of all thread groups in your program. Each thread group expands to show its threads and contains a stack frame trace representing the current method call sequence. Each stack frame expands to show available data elements that are in scope. Icons visually indicate the type of data element. (Static data is not displayed in this view, but is displayed in the Loaded classes and static data view.) Items that are dimmed are inherited.

The default display of this view is split into two panes. The left pane can expand to show stack frames. The right pane displays the content of the item selected on the left, showing anything from a thread group to a variable. For example, if a thread is selected in the left pane, the right pane will show the stack frames for that thread. Alternatively, if a stack frame is selected in the left pane, the right pane will show the variables available in that view. (The split pane is a feature of JBuilder SE and Enterprise.)

For more information about threads, see [“Managing threads” on page 8-30](#).

**Table 8.11** Icons in Threads, call stacks, and data view

Icon	Description
	The current stepping thread.
	A thread group.
(Yellow)	A blocked thread.
	A suspended thread.
(Gray)	A dead thread.
	A class.
	An interface.
(Colored)	An object.
(Shaded)	A null object.
	A stack frame.
	The selected stack frame.
	An array.
	A primitive.
(Red)	An error.
(Gray)	An informational message.

**Table 8.12** Context menu with selection in Threads, call stacks, and data view

Command	Description
Keep Thread Suspended	The selected thread will not be resumed when Run   Resume Program is selected. Allows you to watch behavior of other threads. (JBuilder Enterprise)
Set Stepping Thread	The selected thread will be stepped into when Run   Resume Program is selected. Allows you to watch the behavior of this thread. (JBuilder Enterprise)
Set Execution Point	Sets the stack frame on which resume operations will be performed. (JBuilder Enterprise)
Cut	Removes the value of a variable and puts it in the clipboard. This command is available when a variable is selected. (JBuilder SE and Enterprise)
Copy	Copies the value of a variable to the clipboard. This command is available when a variable is selected. (JBuilder SE and Enterprise)
Paste	Pastes the clipboard contents into another variable. When the Paste command is used, both the cut or copied object variable and the pasted object variable point to the same object. This command is available when a variable is selected. (JBuilder SE and Enterprise)
Create Local Variable Watch	Displays the Add Watch dialog box, where you create a watch on the selected local variable. The watch is added to the Data watches view. This command is available when a variable or variable array is selected. (JBuilder SE and Enterprise)
Create Array Watch	Displays the Add Watch dialog box, where you create a watch on the selected array. The watch is added to the Data watches view. This command is available when an array is selected. (JBuilder SE and Enterprise)
Create Array Component Watch	Displays the Add Watch dialog box, where you create a watch on the selected array component. The watch is added to the Data watches view. This command is available when a component in an array is selected. (JBuilder SE and Enterprise)
Adjust Display Range	Displays the Adjust Range dialog box, where you can adjust the number of array items that are displayed in the view. This command is available when an array is selected. (JBuilder SE and Enterprise)
Create 'this' Watch	Displays the Add Watch dialog box, where you create a watch on the selected <code>this</code> object. The watch is added to the Data watches view. This command is available when a <code>this</code> object is selected. (JBuilder SE and Enterprise)
Create Class Watch	Displays the Add Watch dialog box, where you create a watch on the selected class. The watch is added to the Data watches view. This command is available when a class is selected. (JBuilder SE and Enterprise)


**Table 8.12** Context menu with selection in Threads, call stacks, and data view (continued)

Command	Description
Create Object Watch	Displays the Add Watch dialog box, where you create a watch on the selected object. The watch is added to the Data watches view. This command is available when an object is selected. An object watch watches the selected Java object. It expands to show data members for the current instantiation. (JBuilder SE and Enterprise)
Create String Watch	Displays the Add Watch dialog box, where you create a watch on the selected <code>String</code> . The watch is added to the Data watches view. This command is available when a <code>String</code> is selected. (JBuilder SE and Enterprise)
Create Static Field Watch	Displays the Add Watch dialog box, where you create a watch on the selected static field. The watch is added to the Data watches view. A static field is a Java variable defined as static (a class variable). This command is available when a static field is selected. (JBuilder Enterprise)
Create Field Watch	Creates a watch on the selected field and automatically adds the watch to the Data watches view. A field is a Java variable that is defined in a Java object. This command is available when a field is selected. (JBuilder SE and Enterprise)
Create Field Breakpoint	Creates a breakpoint on the selected field and automatically adds the breakpoint to the Data and code breakpoints view. A field is a Java variable that is defined in a Java object. To activate the breakpoint, go to the Data and code breakpoints view and right-click the breakpoint. Choose Break On Read to force the debugger to stop when the field is about to be read, or Break On Write to stop when the field is about to be written to. This command is available when a field is selected. (JBuilder SE and Enterprise)
Show Hex/Decimal Value	Changes the display base of a numeric or character variable. This command is available when a numeric or character variable is selected. Selecting this command for an array will change the base of its elements.
Show/Hide Null Values	Toggles the display of a null values in an array. This command is useful when debugging a hash-map object. It is available when an array of type <code>Object</code> is selected. (JBuilder SE and Enterprise)
Change Value	Displays the Change Value dialog box, where you can directly edit the value of a variable. This command is available when a variable is selected. (JBuilder SE and Enterprise)

**Table 8.13** Context menu with no selection in Threads, call stacks, and data view






Command	Description
Floating Window	Turns the view into a floating window. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Restore Default View Order	Restores the default order of the debugger views. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Show Current Thread Only	Displays call stacks and data for the current thread only. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Split Threads And Data View	Splits the display into a two-paned view. Left side expands to show stack frames; right side shows the content of the item selected on the left. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)

## Data watches view

 The Data watches view displays the current values of data members that you want to track. You can expand some types of watch expressions to show data elements that are in scope. If elements are not in scope, the message `<not in scope>` will be displayed in the view. Grayed-out items are inherited.

For more information on data watches, see [“Watching expressions” on page 8-59](#).

**Table 8.14** Icons in Data watches view

Icon	Description
 (Colored)	An object.
	An array.
	A primitive.
 (Red)	An error.
 (Gray)	An informational message.

**Table 8.15** Context menu with watch selected in Data watches view

Command	Description
Remove Watch	Removes the selected watch.
Create Local Variable Watch	Displays the Add Watch dialog box, where you create a watch on the selected local variable. This command is available when a variable or variable array is selected. (JBuilder SE and Enterprise)



**Table 8.15** Context menu with watch selected in Data watches view (continued)

Command	Description
Create Array Watch	Displays the Add Watch dialog box, where you create a watch on the selected array. This command is available when an array is selected. (JBuilder SE and Enterprise)
Create Array Component Watch	Displays the Add Watch dialog box, where you create a watch on the selected array component. This command is available when a component of an array is selected. (JBuilder SE and Enterprise)
Adjust Display Range	Displays the Adjust Range dialog box, where you can adjust the number of array items that are displayed in the view. This command is available when an array is selected. (JBuilder SE and Enterprise.)
Create 'this' Watch	Displays the Add Watch dialog box, where you create a watch on the selected <code>this</code> object. The watch is added to the Data watches view. This command is available when a <code>this</code> object is selected. (JBuilder SE and Enterprise)
Create Class Watch	Displays the Add Watch dialog box, where you create a watch on the selected class. The watch is added to the Data watches view. This command is available when a class is selected. (JBuilder SE and Enterprise)
Create Object Watch	Displays the Add Watch dialog box, where you create a watch on the selected object. The watch is added to the Data watches view. This command is available when an object is selected. An object watch watches the selected Java object. It expands to show data members for the current instantiation. (JBuilder SE and Enterprise)
Create String Watch	Displays the Add Watch dialog box, where you create a watch on the selected <code>String</code> . The watch is added to the Data watches view. This command is available when a <code>String</code> is selected. (JBuilder SE and Enterprise)
Create Static Field Watch	Displays the Add Watch dialog box, where you create a watch on the selected static field. The watch is added to the Data watches view. A static field is a Java variable defined as static (a class variable). This command is available when a static field is selected. (JBuilder Enterprise)
Create Field Watch	Creates a watch on the selected field and automatically adds the watch to the Data watches view. A field is a Java variable that is defined in a Java object. This command is available when a field is selected. (JBuilder SE and Enterprise)

**Table 8.15** Context menu with watch selected in Data watches view (continued)

Command	Description
Create Field Breakpoint	Creates a breakpoint on the selected field and automatically adds the breakpoint to the Data and code breakpoints view. A field is a Java variable that is defined in a Java object. To activate the breakpoint, go to the Data and code breakpoints view and right-click the breakpoint. Choose Break On Read to force the debugger to stop when the field is about to be read, or Break On Write to stop when the field is about to be written to. This command is available when a field is selected. (JBuilder SE and Enterprise)
Show/Hide Null Values	Toggles the display of a null values in an array. This command is useful when debugging a hash-map object. It is available when an array of type <code>Object</code> is selected. (JBuilder SE and Enterprise)
Change Value	Displays the Change Value dialog box, where you can directly edit the value of a variable. This command is available when a primitive data type is selected. (JBuilder SE and Enterprise)
Change Watch	Displays the Change Watch dialog box where you change the watch expression and description.

**Table 8.16** Context menu with no selection in Data watches view

Command	Description
Floating Window	Turns the view into a floating window. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Restore Default View Order	Restores the default order of the debugger views. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Add Watch	Displays the Add Watch dialog box, where you can add a watch. This command is available when you right-click an empty area of the view.
Remove All	Removes all watches. This command is available when you right-click an empty area of the view.

## Loaded classes and static data view











The Loaded classes and static data view displays the classes currently loaded by the program. Expanding a class shows static data, if any, for that class. If a package is displayed in the tree, the number of classes loaded for that package is displayed.

Classes in this view that contain \$ followed by a number represent inner classes. Inner classes are created by the compiler for event handlers defined as Anonymous Adapters on the Generated page of the Project Properties dialog box.

For more information, see [“How variables are displayed in the debugger” on page 8-56.](#)

**Table 8.17** Icons in Loaded classes and static data view

Icon	Description
	A package.
	A class.
	An interface.
	A locked class.
 (Colored)	An object.
 (Shaded)	A null object.
	An array.
	A primitive.

**Table 8.18** Context menu with selection in Loaded classes and static data view

Command	Description
Cut	Removes the value of a variable and puts it in the clipboard. This command is available when a variable is selected. (JBuilder SE and Enterprise)
Copy	Copies the value of a variable to the clipboard. This command is available when a variable is selected. (JBuilder SE and Enterprise)
Paste	Pastes the clipboard contents into another variable. When the Paste command is used, both the cut or copied object variable and the pasted object variable point to the same object. This command is available when a variable is selected. (JBuilder SE and Enterprise)
Create Local Variable Watch	Displays the Add Watch dialog box, where you create a watch on the selected local variable. The watch is added to the Data watches view. This command is available when a variable is selected. (JBuilder SE and Enterprise)
Create Array Watch	Displays the Add Watch dialog box, where you create a watch on the selected array. The watch is added to the Data watches view. This command is available when an array is selected. (JBuilder SE and Enterprise)
Create Array Component Watch	Displays the Add Watch dialog box, where you create a watch on the selected array component. The watch is added to the Data watches view. This command is available when a component in an array is selected. (JBuilder SE and Enterprise)
Adjust Display Range	Displays the Adjust Range dialog box, where you can adjust the number of array items that are displayed in the view. This command is available when an array is selected. (JBuilder SE and Enterprise)

**Table 8.18** Context menu with selection in Loaded classes and static data view (continued)

Command	Description
Create 'this' Watch	Displays the Add Watch dialog box, where you create a watch on the selected <code>this</code> object. The watch is added to the Data watches view. This command is available when a <code>this</code> object is selected. (JBuilder SE and Enterprise)
Create Class Watch	Displays the Add Watch dialog box, where you create a watch on the selected class. The watch is added to the Data watches view. This command is available when a class is selected. (JBuilder SE and Enterprise)
Create Object Watch	Displays the Add Watch dialog box, where you create a watch on the selected object. The watch is added to the Data watches view. This command is available when an object is selected. An object watch watches the selected Java object. It expands to show data members for the current instantiation. (JBuilder SE and Enterprise)
Create String Watch	Displays the Add Watch dialog box, where you create a watch on the selected <code>String</code> . The watch is added to the Data watches view. This command is available when a <code>String</code> is selected. (JBuilder SE and Enterprise)
Create Static Field Watch	Displays the Add Watch dialog box, where you create a watch on the selected static field. The watch is added to the Data watches view. A static field is a Java variable that is defined as static in a Java object. This command is available when a static field is selected. (JBuilder Enterprise)
Create Field Watch	Creates a watch on the selected field and automatically adds the watch to the Data watches view. A field is a Java variable that is defined in a Java object. This command is available when a field is selected. (JBuilder SE and Enterprise)
Change Value	Displays the Change Value dialog box, where you can directly edit the value of a variable. This command is available when a variable is selected. (JBuilder SE and Enterprise)

**Table 8.19** Context menu with no selection in Loaded classes and static data view

Command	Description
Floating Window	Turns the view into a floating window. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Restore Default View Order	Restores the default order of the debugger views. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)

## Synchronization monitors view

This is a feature of  
JBuilder SE and  
Enterprise





The Synchronization monitors view shows synchronization monitors used by the threads and their current state, useful for detecting deadlocked situations. In JBuilder Personal, the tab will display, but the deadlock state is not shown.

**Note** Some VMs, such as HotSpot, don't provide this information. If the Synchronization monitors view is not available and your VM supports classic, you need to add `-classic` as a VM parameter as follows:

- 1 Open the Project | Project Properties dialog box for the project you are debugging.
- 2 Choose the Run page, select the runtime configuration, and click Edit to edit it.
- 3 On the Run tab, enter `-classic` in the VM Parameters field.
- 4 Click OK two times to close the dialog boxes.

For more information about threads, see [“Managing threads” on page 8-30](#).

**Table 8.20** Icons in Synchronization monitors view

Icon	Description
 (Yellow)	Synchronization monitor used by specified thread is not locked.
 (Red)	Synchronization monitor used by specified thread is locked.

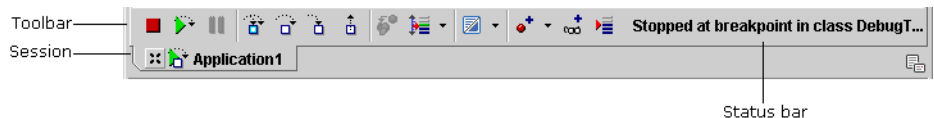
**Table 8.21** Context menu in Synchronization monitors view

Command	Description
Floating Window	Turns the view into a floating window. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)
Restore Default View Order	Restores the default order of the debugger views. This command is available when you right-click an empty area of the view. (JBuilder SE and Enterprise)

## Debugger toolbar


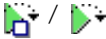











The toolbar at the bottom of the debugger provides quick access to frequently used debugger actions. The right side of the toolbar, the debugger status bar, displays status messages.

**Figure 8.2** Debugger toolbar



The following table explains the toolbar buttons in detail.

**Table 8.22** Toolbar buttons

Button	Action	Description
	Reset Program	Ends the current application run and releases it from memory. This is the same as Run   Reset Program.
	Restart/Resume Program	Restarts the debugging that has finished or been reset or continues the current one. This is the same as Run   Resume Program.
	Pause Program	Pauses the current debugging session. This is the same as Run   Pause Program.
	Smart Step	Controls whether to use the Smart Step settings in the Classes with tracing disabled view and the Smart Step options on the Debug page of the Runtime Properties dialog box. (JBuilder SE and Enterprise)
	Step Over	Steps over the current line of code. This is the same as Run   Step Over.
	Step Into	Steps into the current line of code. This is the same as Run   Step Into.
	Step Out	Steps out of the current method and returns to its caller. This is the same as Run   Step Out.
	Smart Swap	Compiles modified files and updates the compiled classes. This is the same as Run   Smart Swap. (JBuilder Enterprise)
	Set Execution Point	Sets where program is to resume. The execution point is moved to the new location. This is the same as Run   Set Execution Point. (JBuilder Enterprise)
	Smart Source	Sets source file type, based on the original non-Java source language. Positions cursor in new file display on current stack frame. This is the same as Run   Smart Source. (JBuilder Enterprise)
	Add Breakpoint	Adds a breakpoint to the current debugging session. Click the down-facing arrow to the right of the button to choose the breakpoint type. This is the same as Run   Add Breakpoint.
	Add Watch	Adds a watch to the current debugging session. This is the same as Run   Add Watch.
	Show Current Frame	Displays the current thread's call stack and highlights the current execution point in the source.

## Debugger shortcut keys

You can use the following shortcut keys for easy access to debugger functions.

**Table 8.23** Debugger shortcut keys

Keys	Action
<i>Shift+F9</i>	Debug project.
<i>Ctrl+F2</i>	Reset program.
<i>F4</i>	Run to cursor.
<i>F5</i>	Toggle breakpoint when in editor.
<i>F7</i>	Step into.
<i>F8</i>	Step over.
<i>F9</i>	Resume program (continues the current debug session).
<i>Ctrl+left-mouse click</i> in gutter on breakpoint	Displays Breakpoint Properties dialog box.
<i>Ctrl+right-mouse click</i> in editor on expression	Brings up ExpressionInsight window for that expression.

## ExpressionInsight

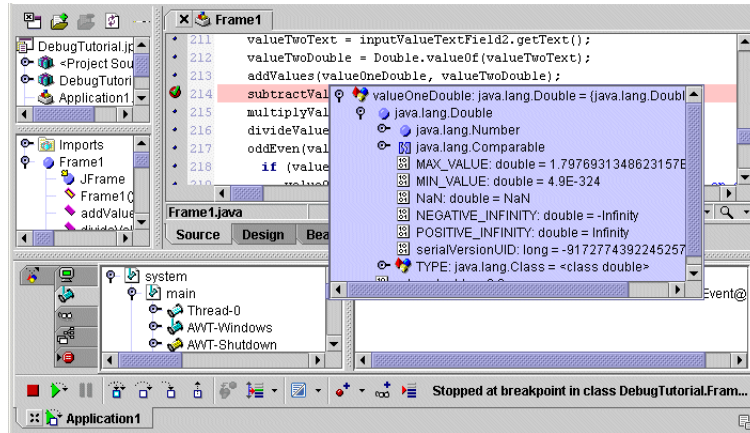
This is a feature of  
JBuilder SE and  
Enterprise

When the debugger is suspended, you can access ExpressionInsight—a small, pop-up window that shows, in tree form, the contents of the selected expression. To display the ExpressionInsight window,

- Hold down the *Ctrl* key (the *Command* key on Macintosh) and move the mouse over your code in the editor. The ExpressionInsight window displays when the mouse passes over a meaningful expression.
- Move your mouse to the expression you want to see in more detail and press *Ctrl* plus the right mouse button.

The ExpressionInsight window displays until you press a key to close it.

The ExpressionInsight window allows you to descend into members of the expression. If the expression is an object, the context menu displays the same menu commands as those available in the Threads, call stacks, and data view when an object is selected. You can also right-click a descendent in the window to display a context menu.

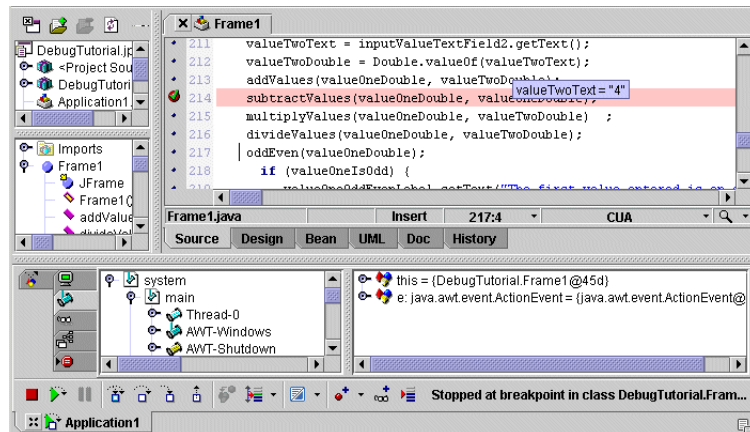
**Figure 8.3** ExpressionInsight window

The ExpressionInsight window is disabled when the debugging session is ended or not suspended.

## Tool tips

This is a feature of  
JBuilder SE and  
Enterprise

When the debugger is suspended, you can place the mouse cursor over any variable in the editor to display its value. The value is displayed in a small pop-up window called a tool tip. If you select text, you'll see the value of the selected text.

**Figure 8.4** Tool tip window

Tool tips are disabled when the debugging session is ended or not suspended.



## Debugging non-Java source

---

This is a feature of  
JBuilder Enterprise

You can use JBuilder to debug non-Java source code, including JSP, SQLJ and LegacyJ code. You can debug both locally and remotely. To accomplish this, JBuilder uses the mapping information that is saved in the class file (see JSR-45). This allows you to debug as you normally would—you can run and suspend your program, set and run to breakpoints, step through code, and examine and change data values.

When your program is suspended, you can change the view of your code, allowing you to view either the Java source code or the non-Java source code. For example, if you're debugging a JSP and you're stopped on a breakpoint, you can either view the Java source for that JSP or the JSP itself.



To switch views, use the Smart Source button on the debugger toolbar. A pop-up window shows the currently selected source view and the available source views. When you select a source view that differs from the current source view, the editor will repaint with the file associated with the new source view. Source view state is kept per debugging session, so changing the source view will change the file that is displayed when the VM is suspended.

**Note** The default source view is the one JBuilder determines is best for the current code.

When you switch source views, the current stack frame and the cursor location will also switch. For example, if you're debugging a JSP, and viewing the JSP code, you might have set the breakpoint on line 25. However, if you switch to the Java source, the cursor might switch to line 75. This is because the analogous stack frames are not located on the same lines of code in the two files.

## Controlling program execution

---

The most important characteristic of a debugger is that it lets you control the execution of your program. For example, you can control whether your program executes a single line of code, an entire method, or an entire program block. By manually controlling when the program should run and when it should pause, you can quickly move over the sections that you know work correctly and concentrate on the sections that are causing problems.

### Running and suspending your program

---

When your program is running in the debugger, you need to pause it in order to examine data values. Pausing causes the debugger to suspend

your program. You can then use the debugger to examine the state of your program with respect to the program location.

When you are using the debugger, your program can be in one of two possible states: *running* or *suspended*.



- Your program is running when the Pause button is available on the debugger toolbar.
- Your program is suspended when you click the Pause icon. When your program is suspended, you can examine data values. The stepping buttons on the debugger toolbar become available.



To resume program execution, choose the Resume Program button on the debugger toolbar. When the debugging session is ended, this button becomes the Restart Program button and restarts the session.



While your program is suspended, you can modify code and resume execution at any active stack frame. For more information, see [“Modifying code while debugging” on page 8-64](#).

## Resetting the program

---

During debugging, you may need to reset the program from the beginning. For example, you might need to reset the program if you step past the location of a bug, or if variables or data structures become corrupted with unwanted values.

To end the current program run, do one of the following:

- Choose Run | Reset Program.
- Click the Reset Program button on the debugger toolbar.



Resetting a program releases resources and clears all variable settings. However, resetting a program does not delete any breakpoints or watches that you have set, which makes it easy to resume the debugging session.



To restart the program, click the Restart Program button on the debugger toolbar.

## The execution point

---

When you're in a suspended debugging session, the line of code that is the current execution point for a thread is highlighted in the editor with an arrow in the left margin of the editor.



The execution point marks the current line of source code to be executed by the debugger. When you pause the program's execution in the debugger, the current execution point for the selected thread is highlighted. The execution point always shows the current line of code to

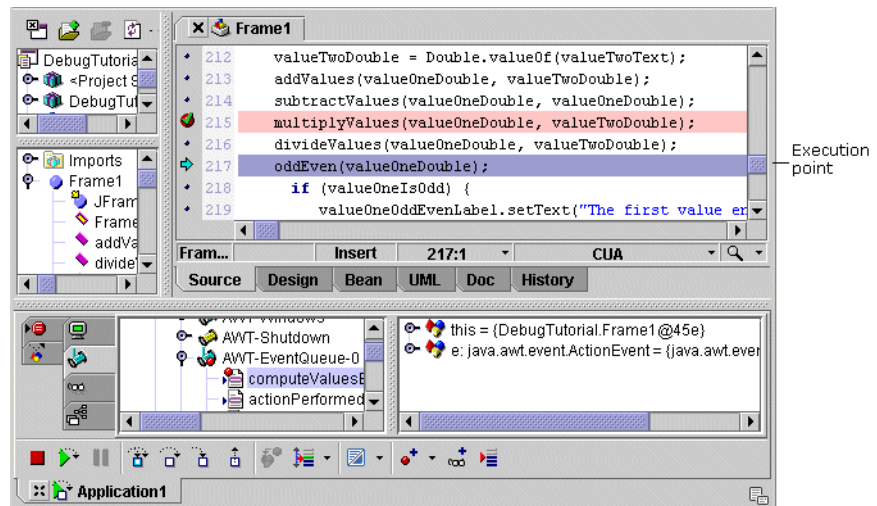
be executed, whether you are going to step over, step into, or run your program without stopping.

To find the current execution point, do one of the following:

- Choose Run | Show Execution Point.
- Click the Show Current Frame button on the debugger toolbar.

The editor displays the block of code in the area of the current execution point. The execution point is marked by an arrow in the left margin of the editor and that line of code is highlighted. Program execution resumes from that point.

**Figure 8.5** The execution point



While debugging, you're free to open, close, and navigate through any file in the editor. Because of this, it's easy to lose track of the next program statement to execute, or the location of the current program scope. To quickly return to the execution point, choose Run | Show Execution Point or click the Show Current Frame button on the debugger toolbar.

## Setting the execution point

This is a feature of  
JBuilder Enterprise

When the program is suspended, you can set the execution point for the current stepping thread. This will change the execution point from its current location. You may also want to set the execution point after you use the Smart Swap button. (For more information about Smart Swap, see ["Modifying code while debugging" on page 8-64.](#))

To set the execution point for the current stepping thread,

- 1 Open the Threads, call stack and data view.
- 2 Select the stack frame where you want to resume operations, right-click and choose Set Execution Point.
- 3 The editor displays source code in the area of the new execution point. If this is the stepping thread, the execution point is highlighted in the left margin of the editor with an arrow and the line of code is highlighted. Program execution resumes from that point.



You can use the Set Execution Point button on the Debugger toolbar to set the execution point. This button displays a pop-up window that lists the stack frames of the stepping thread. You can choose the one you want. Note that the current stack frame selection is dimmed. The stack frame where program execution will resume is marked in the Threads, calls stacks and data view with the stepping button. You can also use the Run | Set Execution Point menu command to set a new stack frame for resume operations.

**Note** Setting the execution point will not reset the value of any object variables that have been modified in the call stacks.

## Managing threads

---

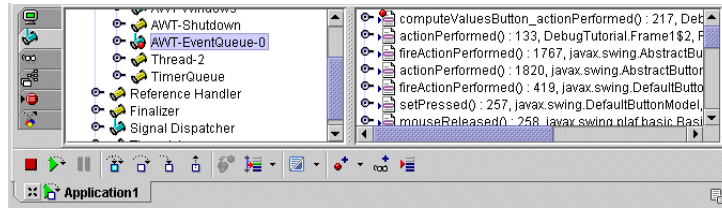
To use the debugger to manage the threads in your program, you use both the Threads, call stacks, and data view and the Synchronization monitors view.

- The Threads, call stacks, and data view shows the current status of all thread groups for the program. It also shows all method calls the program has made, in the order they were called. This display allows you to trace what calls were made to arrive at the current error. You can also use this pane to return to the place where a method was called.
- The Synchronization monitors view shows all the synchronization monitors used by all the threads in the debugged program, and their current state.

### Using the split pane

This is a feature of  
JBuilder SE and  
Enterprise

The default display of the Threads, call stacks, and data view is split into two panes. The left pane can expand to show stack frames. The right pane displays the content of the item selected on the left, showing anything from a thread group to a variable. For example, if a thread is selected in the left pane; the right pane shows the stack frames for that thread. Alternatively, if a stack frame is selected in the left pane, the right pane will show the variables available in that view.

**Figure 8.6** Threads, call stacks, and data view split pane

## Displaying only the current thread

This is a feature of  
JBuilder SE and  
Enterprise

To display the call stacks and data for the current thread only,

- 1 Display the Threads, call stacks, and data view.
- 2 Right-click an empty area of the view.
- 3 Choose Show Current Thread Only. All threads other than the current one are removed from the view.
- 4 To display all threads again, right-click in an empty area of the view and toggle Show Current Thread.


## Displaying the top stack frame



To display the current thread's top stack frame, click the Show Current Frame button on the debugger toolbar.

## Choosing the thread to step into

To choose the thread to step into,

- 1 In the Threads, call stacks, and data view, make sure all threads are showing. (Right-click and make sure Show Current Thread Only is off.)
- 2 Select the thread you want to step into.
- 3 Right-click and choose Set Stepping Thread. The icon for the new stepping thread changes to .

## Keeping a thread suspended

This is a feature of  
JBuilder Enterprise

After the debugger has been suspended, and you're ready to resume execution, you can optionally keep a thread suspended. This allows you to watch the behavior of the just the threads you want, without interference from the others.



To resume program execution, choose the Resume Program button on the debugger toolbar. When the debugging session is resumed, only the threads not kept suspended will be resumed.

**Warning** This can lead to a deadlocked situation.

To keep a thread suspended,



- 1 Start your program and pause the debugger with the Pause button.
- 2 In the Threads, call stacks, and data view, right-click the thread you want to keep suspended.
- 3 Choose Keep Thread Suspended.



- 4 Click the Resume Program button.
- 5 The selected thread will not be resumed.

A suspended thread is indicated by the icon.

## Detecting deadlock states

The ability to detect  
deadlocked threads is a  
feature of JBuilder  
Enterprise

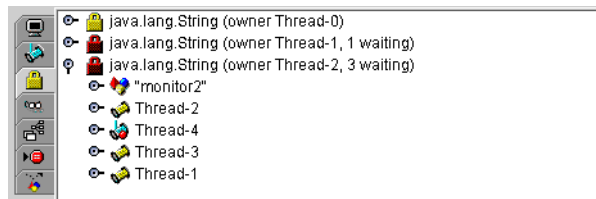
Use the Synchronization monitors view to detect deadlocked threads. You can use this view to see exactly what thread is waiting for what monitor.

In this view, each monitor shows the thread that owns it and the threads, if any, that are waiting to get it. When a monitor is in a deadlocked state, more than one thread is trying to get it. However, it cannot be released, because the thread that is holding it is waiting for another monitor to be released. The (red) icon shows that a monitor is deadlocked.

When you expand each monitor in the view, you'll see all the threads that are waiting for it, as well as the thread that owns it. Note that the icons in this view just show whether that thread is the currently active one. Each thread can be expanded to show its current stack, as it can in the Threads, call stacks, and data view.

In the example below, both threads 1 and 2 have waiting threads. Threads 4, 3, and 1 are waiting for thread 2; thread 2 is the owner—"monitor2" is the name of the thread object. Thread 4 is the current stepping thread.

**Figure 8.7** Synchronization monitors view



## Moving through code

The Run | Step Into and Run | Step Over commands offer the simplest way of moving through your program code. While the two commands are very similar, they each offer a different way to control code execution.

**Note** You can also use the Step Into or Step Over buttons on the debugger toolbar.

The smallest increment by which you step through a program is a single line of code. Multiple program statements on one line of text are treated as a single line of code; you cannot individually debug multiple statements contained on a single line of text. The easiest approach is to put each statement on its own line. A single statement that is spread over several lines of text is treated as a single line of code.

As you debug, you can step into some methods and step over others. If you're confident that a method is working properly, you can step over calls to that method, knowing that the method call will not cause an error. If you aren't sure that a method is well-behaved, step into the method and check whether it is working properly. You should step over methods that are in libraries provided by JBuilder or third party vendors. This will considerably speed up your debugging cycle.

### Stepping into a method call

The Run | Step Into command executes a single program statement at a time. If Smart Step is on, classes in the Classes with tracing disabled view that are marked as tracing disabled will not be stepped into. When Smart Step is off, classes in the Classes with tracing disabled view are ignored, so you'll be able to step into all of these classes.

If the execution point is located on a call to a method, the Step Into command steps into that method and places the execution point on the method's first statement. Subsequent Step Into commands will execute the method's code one line at a time.

If the execution point is located on the last statement of a method, Step Into causes the debugger to return from the method, placing the execution point on the line of code that follows the call to the method you are returning from.

The term "single-stepping" refers to using Step Into to successively run though the statements in your program code.

There are several ways to issue the Step Into command:

- Choose Run | Step Into.
- Press *F7*.
- Click the Step Into button on the debugger toolbar.



### Stepping over a method call

The Run | Step Over command, like Run | Step Into, lets you execute program statements one at a time. However, if you issue the Step Over command when the execution point is located on a method call, the debugger runs that method without stopping (instead of stepping into it), then positions the execution point on the statement that follows the method call.

There are several ways to issue the Step Over command:

- Choose Run | Step Over.
- Press *F8*.
- Click the Step Over button on the debugger toolbar.



## Stepping out of a method

The Run | Step Out command lets you step out of a method to the calling routine.

If Smart Step is on, classes in the Classes with tracing disabled view that are marked as tracing disabled will not be stopped in.

There are two ways to issue the Step Out command:

- Choose Run | Step Out.
- Click the Step Out button on the debugger toolbar.



## Using Smart Step

The Smart Step toggle allows you to determine if each step is “smart” or not. To set this toggle, choose Enable Smart Step on the Debug page of the Runtime Properties dialog box. You can also click the Smart Step button on the debugger toolbar to enable Smart Step for the current session.



When this feature is on, each step operation steps into the classes listed in the Classes with tracing disabled view. For JBuilder SE and Enterprise, stepping is also controlled by the Smart Step options on the Debug page of the Runtime Properties dialog box.

- The Classes with tracing disabled view allows you to set what classes won’t be traced into. For JBuilder Personal, just three classes are available in this view: `java.lang.Object`, `java.lang.String` and `java.lang.ClassLoader`. You cannot add, modify, or delete items in the view.
- For JBuilder SE and Enterprise, the Smart Step options on the Debug page of the Runtime Properties dialog box control the stepping behavior for the classes that are traced into. These options are:
  - Skip synthetic methods  
Skips synthetic methods when stepping into classes.
  - Skip constructors  
Skips constructors when stepping into classes.
  - Skip static initializers  
Skips static initializers when stepping into classes.



- Warn if break in class with tracing disabled (This is a feature of all JBuilder editions.)

Displays a warning message if there is a breakpoint in a class that has tracing disabled. For more information, see [“Breakpoints and tracing disabled settings” on page 8-40](#).

When Smart Step is off, classes in the Classes with tracing disabled view, along with Smart Step options, are ignored, so you’ll be able to step into all of these classes.

By default, when you start a debugging session, Smart Step is on.



- To turn it off for the current session, click the Smart Step button on the debugger toolbar. To turn it off at the start of a debugging session, turn off the Enable Smart Step option on the Debug page of the Runtime Properties dialog box.
- The Smart Step button on the debugger toolbar dims to show that Smart Step is off. To turn Smart Step back on again, click the button or set the Enable Smart Step option on the Debug page of the Runtime Properties dialog box.

## Running to a breakpoint

---

Set breakpoints on lines of source code where you want the program execution to pause during a run. Running to a breakpoint is similar to running to a cursor position, in that the program runs without stopping until it reaches a certain source code location.

You can have multiple breakpoints in your code. You can customize each breakpoint so it pauses the program execution only when certain conditions occur.



If you are debugging non-Java source and your program is paused on a breakpoint, you can switch views. Press the Smart Source button on the debugger toolbar, or choose Run | Smart Source. Choose the view of your code that you want to see. The source is displayed in the editor, and the cursor is placed on the current stack frame. Note that this will probably be different line number than in the previous view. For example, if you’re debugging a JSP, you might be on line 120 in the Java source, but on line 55 in the JSP source (the non-Java source). For more information, see [“Debugging non-Java source” on page 8-27](#).

For more information about breakpoints, see [“Using breakpoints” on page 8-40](#).

## Running to the end of a method

---

The Run | Run To End Of Method command runs your application until it reaches the end of the current method. This command is useful if you've stepped into a method you meant to step over.

## Running to the cursor location

---

You can run your program to a spot just before the suspected location of the problem. At that point, check that all data values are correct. Then run your program to another location, and check the values there.

To run to a specific program location,

- 1 In the editor, position the cursor on the line of code where you want to begin (or resume) debugging.
- 2 Choose Run | Run To Cursor or right-click and choose Run To Cursor.

When you run to the cursor, your program executes without stopping until the execution reaches the location marked by the cursor in the editor. When the execution encounters the code marked by the cursor, the debugger regains control, suspends your program, and places the execution point on that line of code. For more information about the execution point, see [“The execution point” on page 8-28](#).

This command speeds up the debugging process, as it allows you to move quickly through code that is error-free.

## Viewing method calls

---

The Threads, call stacks, and data view shows all thread groups from your program. For each thread, the sequence of method calls that brought your program to its current state is displayed. Each stack frame expands to show available data.

If your program was compiled with debugging information (the default), this view also shows the arguments passed to a method call. Each method is followed by a listing that details the parameters with which the call was made. In addition, the view shows where each method resides. It lists the line the method call is on, the class name, and the source name.

To view the source code and data state located at a particular method call, click the method.

## Locating a method call

You can locate the place in your source code where a method was called, allowing you to backtrack into a debugging session.

To locate a method call, do one of the following:

- Click the method in the Threads, call stack and data pane. This takes you to the editor, with the cursor placed on the line of code in the file from which the method was called.
- Right-click in the editor and choose the Run To Cursor command.

## Controlling which classes to trace into

To closely examine part of your program, you can tell the debugger to only trace into the files you want to step through. This way, you can concentrate on a known problem area, rather than manually stepping through every line of code in the entire program. For example, you usually don't want to step through classes that are in the Sun library, because you're not going to troubleshoot them; you usually only want to inspect and troubleshoot your own classes.

To determine what classes are and aren't going to be traced into,

- If you're in a debugging session, choose the Classes with tracing disabled view. This shows all the classes not to trace into.





- If you haven't started a debugging session, choose Run | View Classes With Tracing Disabled. The Classes With Tracing Disabled dialog box is displayed.



Both the view and the dialog box operate in the same way.

**Note** In JBuilder Personal, three basic classes (`java.lang.Object`, `java.lang.String` and `java.lang.ClassLoader`) are added to the view. You cannot add, modify or delete items in the list; however, you can choose to step or not step into those classes. See “Using Smart Step” on page 8-34 for more information.

You can enable or disable a class or package in the Classes with tracing disabled view at any time. Simply right-click the class or package and toggle the Step Into Class/Package option. When disabled (the default), the class or package won’t be traced into. When enabled, it will be stepped into. Note that the icon changes for the two states:

- When tracing is enabled, the icon is:  (Colored)
- When disabled, the icon is:  (Gray)

**Note** When you disable tracing for a package, you are disabling tracing for all classes in that package.

In JBuilder SE and Enterprise, you can remove a class or package from the list by selecting it and pressing *Delete*, or by selecting it, right-clicking, and choosing Remove Class/Package. To remove all classes and packages, right-click in an empty area of the view and choose Remove All.

Removing all classes and packages from the view automatically enables tracing into every class that your program calls.

In JBuilder SE and Enterprise, you can add a class or a package to the list by right-clicking in an empty area of the view and choosing Add Class Or Package. The Select Class Or Package dialog appears, where you choose the name of the class or package to disable tracing for.

In JBuilder SE and Enterprise, you can edit a class or a package in the list by right-clicking a class or package in the view and choosing Edit Class/Package. The Select Class Or Package dialog appears, where you choose the name of the class or package to enable tracing for.

Changes take place immediately. You do not need to restart the debugging session.

Classes in the Classes with tracing disabled view, with their enabled/disabled state, are saved in the project file.



Once you’ve selected the classes you don’t want to trace into, use the Smart Step button on the debugger toolbar to control the stepping. When this feature is on, each step operation uses the classes listed in the Classes with tracing disabled view and the Smart Step options selected on the Debug page of the Runtime Configuration Properties dialog box:

- The Classes with tracing disabled view allows you to set which classes won’t be traced into.
- The Smart Step options on the Debug page of the Runtime Properties dialog box control the stepping behavior for the classes that are traced into.

When Smart Step is off, classes in the Classes with tracing disabled view, along with the Smart Step options, are ignored, so you'll be able to step into all of these classes.

## Tracing into classes with no source available

This is a feature of  
JBuilder SE and  
Enterprise

If you turn Smart Step off when you're using a class but don't have its source file available, a stub source file is generated and appears as you trace through your code. The stub source file shows only the method signatures for the class. To avoid seeing stub source, keep the class in the Classes with tracing disabled view and leave Smart Step on.

## Stub source files

If stub source is generated for files for which you have source available, check the source path. The debugger looks in your source path for source files. The source path is described in ["Source path" on page 4-10](#). The .java file being debugged has to exist in a branch that is the same as its package name.

For example, if your source path contains one item:

```
c:\MyProjects\Test\src
```

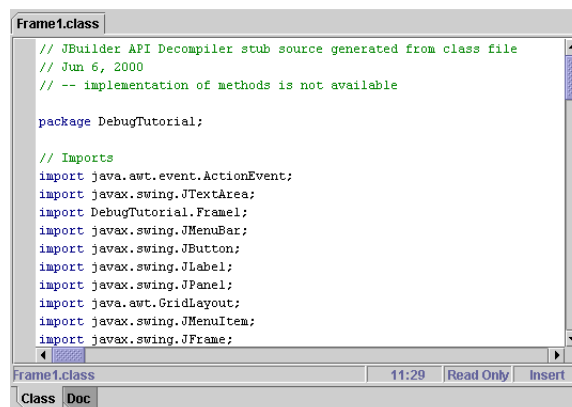
and your .java file is in a package called `mypackage`, the debugger expects the .java file to exist in the directory:

```
c:\MyProjects\Test\src\mypackage
```

The package name is appended to the source item name. If you have multiple source items, the debugger will try to locate all of them using the scheme outlined above. If the debugger can't locate the source file, it generates stub source.

A stub source file is displayed in the content pane. It contains a header and method stubs.

**Figure 8.8** Stub source file



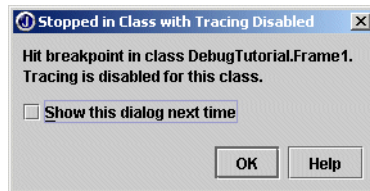
## Breakpoints and tracing disabled settings

Setting a breakpoint in a class in the Classes with tracing disabled view overrides tracing settings; you will pause in the class because you explicitly instructed the debugger to go to that point.

A warning dialog box, the Stopped In Class With Tracing Disabled dialog box, will be displayed if:

- Smart Step is on, and
- The Warn If Break In Class With Tracing Disabled option is on in the Debug page of the Runtime Properties dialog box.

**Figure 8.9** Stopped In Class With Tracing Disabled dialog box



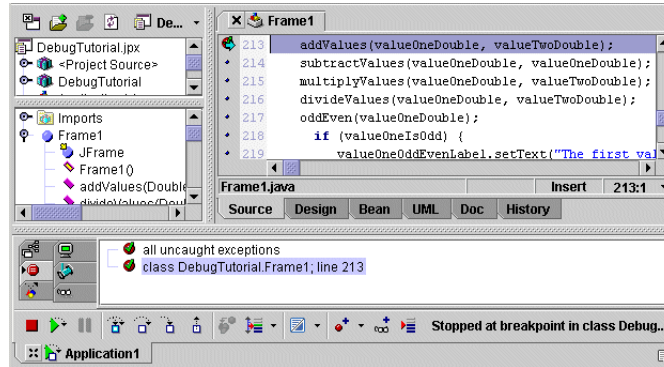
In this situation, stepping after the breakpoint is hit will cause the debugger to go out of that class. To stay in that class, turn Smart Step off, then use the stepping buttons.

## Using breakpoints

---

When your program execution encounters a breakpoint, the program is suspended, and the debugger displays the line containing the breakpoint in the editor. You can then use the debugger to view the state of your program. Breakpoints are flexible in that they can be set before you begin a program run or at any time that the debugger has control. By setting breakpoints in potential problem areas of your source code, you can run your program without pausing until the program's execution reaches a location you want to debug.

Breakpoints are displayed and manipulated in the Data and code breakpoints view. The type of breakpoint and its status are displayed, along with information specific to the breakpoint type, such as line number, class name or method name. You can use the right-click menu to enable and disable breakpoints, as well as add and remove them.

**Figure 8.10** Data and code breakpoints view

## Setting breakpoints

Class, method, exception, and field breakpoints are features of JBuilder SE and Enterprise.

Cross-process breakpoints are a feature of JBuilder Enterprise.


You can set line, exception, class, method, field, and cross-process breakpoints in the debugger:


- A line breakpoint is set on a specific line of Java or non-Java source code. The debugger stops on that line.
- An exception breakpoint causes the debugger to stop when the specified exception is about to be thrown.
- A class breakpoint causes the debugger to stop when any method from the specified class is called or when the specified class is instantiated.
- A method breakpoint causes the debugger to stop when the specified method in the specified class is called.
- A field breakpoint causes the debugger to stop when the specified field is about to be read or written to. A field is a Java variable that is defined in a Java object.
- A cross-process breakpoint causes the debugger to stop when either any method or the specified method in the specified class in a separate process are stepped into.

### Setting a line breakpoint

A line breakpoint causes the debugger to stop when it reaches that particular line of code. Line breakpoints can be set in either Java or non-Java source code. You can set a line breakpoint directly in the editor or use the Add Line Breakpoint dialog box.

To set a line breakpoint in source code, click the left margin of the line you want to set the breakpoint on. You can also press *F5* when on a line of source code to toggle a line breakpoint. When the debugger has focus,

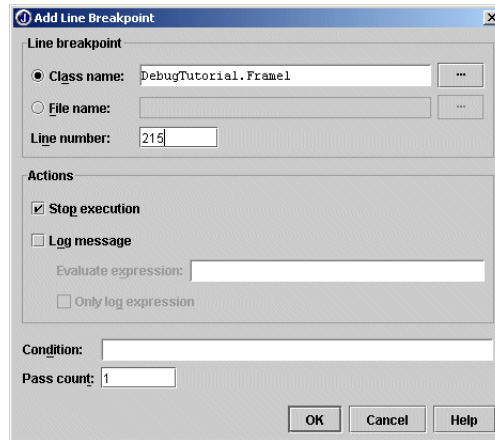
small blue dots  are displayed in the editor to the left of lines of executable code, indicating that a breakpoint can be set on that line.

Breakpoints set on comment lines, declarations, or other non-executable lines of code are invalid. Invalid breakpoints are indicated by  in the gutter of the editor when you run your program.

To set a line breakpoint using the Add Line Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run | Add Breakpoint and choose Add Line Breakpoint.
- When you're in a debugging session, click the down-facing arrow to the right of the Add Breakpoint button on the debugger toolbar and choose Add Line Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Line Breakpoint.

The Add Line Breakpoint dialog box is displayed.



To set a line breakpoint, choose the following options:

- 1 If you're setting a breakpoint in a Java `.class` file, use the Class Name field. If the breakpoint is in a file that is not a `.class` file, use the File Name field.
  - If the file is a `.class` file, either enter the name or choose the ellipsis (...) button to browse to a `.class` file.
  - If the file is not a `.class` file, choose the ellipsis (...) button to browse to the file.



- 2 In the Line Number field, enter the number of the line to set the breakpoint on.
- 3 Choose the Actions for the breakpoint. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression. For more information, see [“Setting breakpoint actions” on page 8-51](#). (Actions are a feature of JBuilder SE and Enterprise.)
- 4 In the Condition field, set the breakpoint condition, if one exists, for this breakpoint. For more information, see [“Creating conditional breakpoints” on page 8-52](#).
- 5 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated. For more information, see [“Using pass count breakpoints” on page 8-53](#).
- 6 Click OK to close the dialog box.



If the breakpoint is valid (set on an executable line of code), the line on which the breakpoint is set becomes highlighted, and a red circle icon with a checkmark appears in the left margin of the breakpointed line.

## Setting an exception breakpoint

Exception breakpoints  
are features of JBuilder  
SE and Enterprise

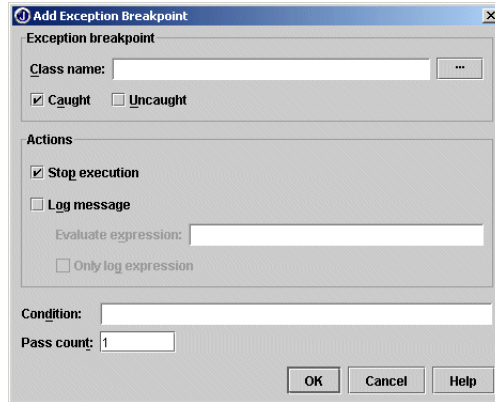
An exception breakpoint causes the debugger to stop when the specified exception is about to be thrown. The debugger can stop on caught and/or uncaught exceptions. To set an exception breakpoint, use the Add Exception Breakpoint dialog box.

To open the Add Exception Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run | Add Breakpoint and choose Add Exception Breakpoint.
- When you're in a debugging session, click the down-facing arrow to the right of the Add Breakpoint button on the debugger toolbar and choose Add Exception Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Exception Breakpoint.



The Add Exception Breakpoint dialog box is displayed.



To set an exception breakpoint,

- 1 Enter the name of the exception class file on which the debugger will stop in the Class Name field. You can either enter the name or choose the ellipsis (...) button to browse to a .class file.
- 2 Choose when the debugger should stop:
  - Select the Caught option to force the debugger to stop when the exception is caught.
  - Select the Uncaught option to force the debugger to stop when the exception is not caught.

You can also choose both Caught and Uncaught to force the debugger to stop in both cases.

- 3 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression. For more information, see [“Setting breakpoint actions” on page 8-51](#). (Actions are a feature of JBuilder SE and Enterprise.)
- 4 In the Condition field, set the condition, if one exists, for this breakpoint. For more information, see [“Creating conditional breakpoints” on page 8-52](#).
- 5 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated. For more information, see [“Using pass count breakpoints” on page 8-53](#).
- 6 Click OK to close the dialog box.

Class breakpoints are  
features of JBuilder SE  
and Enterprise

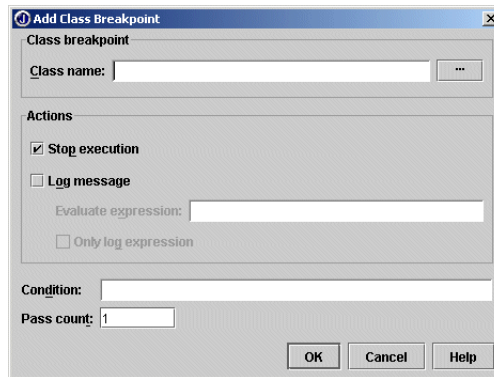
## Setting a class breakpoint

A class breakpoint causes the debugger to stop when any method from the specified class is called or when the specified class is instantiated. To set a class breakpoint, use the Add Class Breakpoint dialog box.

To open the Add Class Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select **Run | Add Breakpoint** and choose **Add Class Breakpoint**.
- When you're in a debugging session, click the down-facing arrow to the right of the **Add Breakpoint** button on the debugger toolbar and choose **Add Class Breakpoint**.
- When you're in a debugging session, right-click an empty area of the **Data and code breakpoints** view and choose **Add Class Breakpoint**.

The Add Class Breakpoint dialog box is displayed.



To set a class breakpoint,

- 1 Enter the name of the class file you want the debugger to stop on in the **Class Name** field. You can either enter the name or choose the ellipsis (...) button to browse to a .class file.
- 2 Choose the **Actions** for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression. For more information, see [“Setting breakpoint actions” on page 8-51](#). (Actions are a feature of JBuilder SE and Enterprise.)
- 3 In the **Condition** field, set the condition, if one exists, for this breakpoint. For more information, see [“Creating conditional breakpoints” on page 8-52](#).
- 4 In the **Pass Count** field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated. For more information, see [“Using pass count breakpoints” on page 8-53](#).
- 5 Click **OK** to close the dialog box.

Method breakpoints are features of JBuilder SE and Enterprise

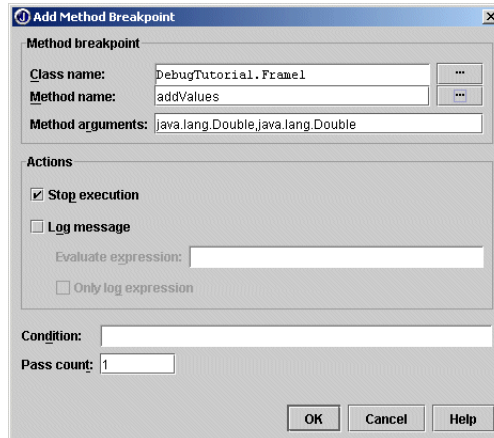
## Setting a method breakpoint

A method breakpoint causes the debugger to stop when the specified method in the specified class is called. To set a method breakpoint, use the Add Method Breakpoint dialog box.

To open the Add Method Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select **Run | Add Breakpoint** and choose **Add Method Breakpoint**.
- When you're in a debugging session, click the down-facing arrow to the right of the **Add Breakpoint** button on the debugger toolbar and choose **Add Method Breakpoint**.
- When you're in a debugging session, right-click an empty area of the **Data and code breakpoints** view and choose **Add Method Breakpoint**.

The Add Method Breakpoint dialog box is displayed.



To set a method breakpoint,

- 1 Enter the name of the class that contains the method you want the debugger to stop on in the **Class Name** field. You can either enter the name or choose the ellipsis (...) button to browse to a .class file.
- 2 In the **Method** field, enter the name of the method you want the debugger to stop on. Click the **Method** button to browse to the method you want.
- 3 In the **Method Arguments** field, enter a comma-delimited list of method arguments. This causes the debugger to stop only when the method name and argument list match. This is useful for overloaded methods. If you used the browser to add the method, method arguments are automatically filled in. If you don't specify any arguments, the debugger stops at all methods with the specified method name.

- 4 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression. For more information, see [“Setting breakpoint actions” on page 8-51](#). (Actions are a feature of JBuilder SE and Enterprise.)
- 5 In the Condition field, set the condition, if one exists, for this breakpoint. For more information, see [“Creating conditional breakpoints” on page 8-52](#).
- 6 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated. For more information, see [“Using pass count breakpoints” on page 8-53](#).
- 7 Click OK to close the dialog box.


## Setting a field breakpoint

Field breakpoints are features of JBuilder SE and Enterprise

A field breakpoint causes the debugger to stop when the specified field is about to be read or written to, depending on your choices. A field is a Java variable that is defined in a Java object. In the following example:

```
class Test {
    private int x;
    private Object y;
}
Test myTest = new Test();
```

`myTest` is a variable. The Java variables `x` and `y` are fields.

To add a field breakpoint, right-click a field variable in the Threads, call stacks, and data view and choose Add Field Breakpoint. The breakpoint is automatically added to the Data and code breakpoints view. A field breakpoint is indicated with .

To control whether the debugger breaks on a read or a write action, open the Data and code breakpoints view. Right-click the field breakpoint you just set. By default, the Break On Read and Break On Write commands in the context menu are enabled, meaning that the debugger will stop when the specified field is about to be read or written to. You can turn off one or both of these options, allowing the debugger to continue, instead of stop, when the field is about to be read or written to.

## Setting a cross-process breakpoint

Cross-process breakpoints are features of JBuilder Enterprise

A cross-process breakpoint causes the debugger to stop when you step into any method or the specified method in the specified class in a separate process. This allows you to step into a server process from a client process, rather than having to set breakpoints on the client side and on the server side. You will usually set a line breakpoint on the client side and a cross-process breakpoint on the server side. For a tutorial that walks through cross-process stepping, see [Chapter 19, “Tutorial: Remote debugging.”](#)

To activate a cross-process breakpoint set on a server process,

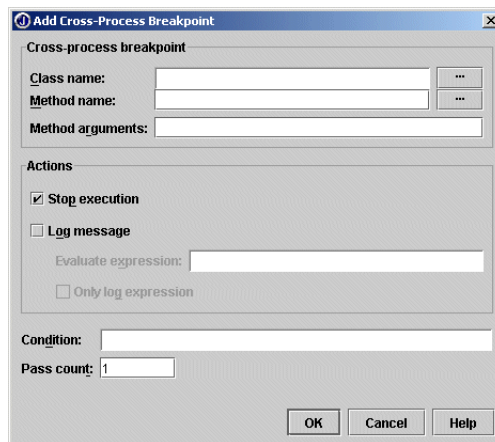
- 1 Start the server process on the remote computer in debug mode.
- 2 On the client computer, from within JBuilder, attach to the server already running on the remote computer.
- 3 Set a line breakpoint in the client code and start debugging the client. At the breakpoint, step into the server code. Do not use Step Over—stepping over will not stop at the cross-process breakpoint.

**Note** You can use cross-process breakpoints to debug locally, for example a client/server application running on one computer.

To set a cross-process breakpoint, use the Add Cross-Process Breakpoint dialog box. To open the Add Cross-Process Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run | Add Breakpoint and choose Add Cross-Process Breakpoint.
- When you're in a debugging session, click the down-facing arrow to the right of the Add Breakpoint button on the debugger toolbar and choose Add Cross-Process Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Cross-Process Breakpoint.

The Add Cross-Process Breakpoint dialog box is displayed.



For a tutorial that explains cross-process stepping, see [Chapter 19, "Tutorial: Remote debugging."](#)

To set a cross-process breakpoint,

- 1 Enter the name of the server-side class that contains the method you want the debugger to stop on in the Class Name field. You can either enter the name or choose the ellipsis (...) button to browse to a `.class` file.
- 2 In the Method field, enter the name of the method you want the debugger to stop on. Use the ellipsis (...) button to display the Select Method dialog box where you can browse through the methods available in the selected class.



The method name is not required. If you do not specify the method name, the debugger stops at all method calls in the specified class.

**Note** You cannot select a method if the selected class contains syntax or compiler errors.

- 3 In the Method Arguments field, enter a comma-delimited list of method arguments. This causes the debugger to stop when the method name and argument list match. This is useful for overloaded methods.
  - If you don't specify any arguments, the debugger stops at all methods with the specified method name.
  - If you choose a method name from the Select Method dialog box, the Methods Argument field is automatically filled in.
- 4 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression. For more information, see [“Setting breakpoint actions” on page 8-51](#). (Actions are a feature of JBuilder SE and Enterprise.)
- 5 In the Condition field, set the condition, if one exists, for this breakpoint. For more information, see [“Creating conditional breakpoints” on page 8-52](#).
- 6 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated. For more information, see [“Using pass count breakpoints” on page 8-53](#).
- 7 Click OK to close the dialog box.

- 8 Set a line breakpoint in the client on the method that calls the cross-process breakpoint.



- 9 Click the Step Into button on the debugger toolbar to step into the server-side breakpointed method. (If you use Step Over, the debugger will not stop.)

## Setting breakpoint properties

---

Once you've created a breakpoint, you can set or change its properties. To set breakpoint properties,

- 1 Open the Data and code breakpoints view.
- 2 Choose the breakpoint you want to set properties for. Right-click and choose Breakpoint Properties.

The Breakpoint Properties dialog box is displayed.

**Note** The Breakpoint Properties dialog box contains the same options as the dialog box you used to create the breakpoint.

- 3 You can change the following properties:

- Actions

The actions to be performed when the breakpoint is hit. The debugger can stop execution at the breakpoint, display a message or evaluate an expression. For more information, see [“Setting breakpoint actions” on page 8-51](#).

- Condition

The condition, if one exists, for this breakpoint. For more information, see [“Creating conditional breakpoints” on page 8-52](#).

- Pass Count

The number of times this breakpoint must be passed in order for the breakpoint to be activated. For more information, see [“Using pass count breakpoints” on page 8-53](#).

To display the Breakpoint Properties dialog box in read-only mode, position the mouse in the gutter next to the breakpointed line. Press *Ctrl* plus the right mouse button. The core properties for the breakpoint are displayed, but they are read-only. You can edit the Actions, Conditions and Pass Count fields.



## Setting breakpoint actions

This is a feature of  
JBuilder SE and  
Enterprise

You can select one more actions to be performed when a breakpoint occurs. The debugger can:

- Stop program execution and display a message in the debugger status bar (the default).
- Log a message in the Console output, input, and errors view.
- Evaluate an expression and log the results.

Actions are defined in the middle area of the Breakpoints dialog box.

**Figure 8.11** Breakpoint actions



### Stopping program execution

To stop program execution when the specified breakpoint is hit, choose the Stop Execution option. If you stop program execution, the debugger will stop at the specified breakpoint, display a status message on the debugger toolbar, and display the breakpoint in the Data and code breakpoints window.

The exact message displayed in the status bar depends on the type of breakpoint. The following example shows the status bar message for a line breakpoint.

**Figure 8.12** Breakpoint status bar message



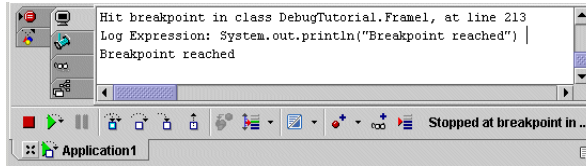
### Logging a message

To log a message to the Console output, input and errors view when the breakpoint is hit, choose the Log Message option and enter a message in the Evaluate Expression box. When the debugger reaches the selected breakpoint, a message will be logged to the view. If the Stop Execution option is also selected the program will stop. Otherwise, it will continue to run.

### Logging a message with a println statement

You can use `println` statements to log output messages. The following example shows a message in the Code and data breakpoints view that was

logged with the statement - `System.out.println("Breakpoint reached")`. The statement was entered in the Evaluate Expression input box and the Log Message option was checked.

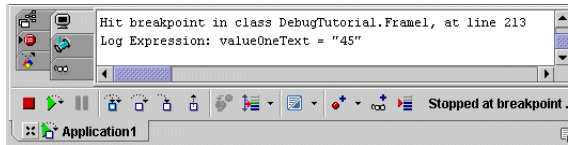


## Logging a message with expression evaluation

You can also use expression evaluation, instead of `println` statements, to log messages while debugging. To do this, enter an expression in the Evaluate Expression input field. The debugger will evaluate this expression when the breakpoint is hit and write the results of the evaluation to the Console output, input, and errors view. The expression can be any valid Java language statement.

This option is available only if you select the Log Message option. Note that you can choose to stop program execution when an expression is evaluated, by choosing the Stop Execution option.

The following example shows the results of an expression `valueOneText`. The expression was entered in the Evaluate Expression field, with the Log Message option checked.




The Only Log Message option allows you to log only the results of this expression, so that the log is not cluttered with other information.

## Creating conditional breakpoints

When a breakpoint is first set, by default it suspends the program execution each time the breakpoint is encountered. However, the Breakpoint Properties dialog box allows you to customize breakpoints so that they are activated only in certain conditions.

By entering a boolean expression in the Condition field, you can make a breakpoint conditional—program execution will stop at this breakpoint only if the condition evaluates to `true`. You can also base a breakpoint on a pass count, specified in the Pass Count field. This field is useful for debugging loops. Program execution stops at the breakpoint after it passes the loop the specified number of times.

Conditions are defined at the bottom of the Breakpoints dialog box.

**Figure 8.13** Conditional breakpoints


The image shows a dialog box with two input fields. The first field is labeled 'Condition:' and is empty. The second field is labeled 'Pass count:' and contains the number '1'.

## Setting the breakpoint condition

The Condition edit box in the Breakpoint Properties dialog box lets you enter an expression that is evaluated each time the breakpoint is encountered during the program execution.

- If the condition evaluates to `true`, the debugger stops at the breakpoint location if the Stop Execution option is on.
- If the condition evaluates to `false`, the debugger doesn't stop at the breakpoint location.

Conditional breakpoints let you see how your program behaves when a variable falls into a certain range or what happens when a particular flag is set.

For example, suppose you want a breakpoint to suspend execution on a line of code only when the variable `mediumCount` is greater than 10. To do so,

- 1 Set a breakpoint on a line of code, by clicking to the left of the line in the editor.
- 2 Right-click and choose Breakpoint Properties.
- 3 Enter the following expression into the Condition edit box, and click OK:

```
mediumCount > 10
```

You can enter any valid Java language expression into the Condition edit box, but all symbols in the expression must be accessible from the breakpoint's location.

## Using pass count breakpoints

The Pass Count condition in the Breakpoint Properties dialog box specifies the number of times that a breakpoint must be passed in order for the breakpoint to be activated. The debugger suspends the program the `n`th time that the breakpoint is encountered during the program run. The default value of `n` is 1.

Pass counts are useful when you think that a loop is failing on the `n`th iteration. When pass counts are used with boolean conditions, program execution is suspended the `n`th time the condition is `true`.

## Disabling and enabling breakpoints

---

Disabling a breakpoint hides it from the current program run. When you disable a breakpoint, all the breakpoint settings remain defined, but the breakpoint is hidden from the execution of your program—your program will not stop on a disabled breakpoint. Disabling a breakpoint is useful if you have defined a conditional breakpoint that you don't need to use now but might need to use at a later time:

- To disable a single breakpoint, right-click it in the Data and code breakpoints view. Choose Enable Breakpoint to toggle the command and disable the breakpoint. When this toggle is off, the breakpoint is disabled and will not be stopped at. When it is on, the breakpoint is enabled and will be stopped at. Breakpoints are enabled by default.
- To disable or enable all breakpoints set for a debugging session, open the Data and code breakpoints view. Right-click an empty area of the view and choose Disable All or Enable All.

You can also disable breakpoints for a runtime configuration. If two or more runtime configurations are defined for the current project, the menu command Disable For Configuration will be available when you right-click a breakpoint. You choose the configuration(s) to disable this breakpoint for. When you debug using the specified configuration, the selected breakpoint will be disabled.

## Deleting breakpoints

---

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the debugging session. You can delete a line breakpoint in the editor. Delete other types of breakpoints with the Data and code breakpoints view.

Note that you cannot delete the default breakpoint, `all uncaught exceptions`. You can, however, disable it.

Use any of the following methods to delete breakpoints:

- In the editor, place the cursor in the line containing the breakpoint, press *F5* or right-click and choose Toggle Breakpoint.
- From the Data and code breakpoints view, highlight the breakpoint you want removed, right-click, and choose Remove Breakpoint or press *Delete*.
- To delete all breakpoints set for a debugging session, open the Data and code breakpoints view. Right-click an empty area of the view and choose Remove All.
- Select a group of breakpoints in the Data and code breakpoints view and press *Delete*.

**Warning** The breakpoint delete commands are not reversible.

## Locating line breakpoints

---

If a line breakpoint isn't displayed in the editor, you can use the Data and code breakpoints view to quickly find the breakpoint's location in your source code.

To locate a line breakpoint,

- 1 In the Data and code breakpoints view, select a line breakpoint.
- 2 Right-click, and select Go To Breakpoint. You can also double-click the selected breakpoint.

The editor shows the breakpoint's location.

## Examining program data values

---

Even though you can discover many interesting things about your program by running and stepping through it, you usually need to examine the values of program variables to uncover bugs. For example, it's helpful to know the value of the index variable as you step through a `for` loop, or the values of the parameters passed in a method call.

When you pause your program while debugging, you can examine the values of instance variables, local variables, properties, method parameters, and array items.

Data evaluation occurs at the level of expressions. An expression consists of constants, variables, and values in data structures, possibly combined with language operators. In fact, almost anything you use on the right side of an assignment operator can be used as a debugging expression.

JBuilder has several features enabling you to view the state of your program, which are described in the table below.

**Table 8.24** Debugger features

Feature	Enables
Loaded classes and static data view	Viewing classes currently loaded by the program, and the static data, if any, for those classes.
Threads, call stack and data view	Viewing the thread groups in your program. Each thread group expands to show its threads and contains a stack frame trace representing the current method call sequence. Each stack frame can expand to show data elements that are in scope.
Data watches view	Viewing the current values of variables that you want to track. A watch evaluates an expression according to the current context. If you move to a new context, the expression is re-evaluated for the new context. If it is no longer in scope, it cannot be evaluated.

**Table 8.24** Debugger features (continued)

Feature	Enables
Evaluate/Modify dialog box	Evaluating expressions, method calls, and variables.
ExpressionInsight	Viewing the values of expressions.

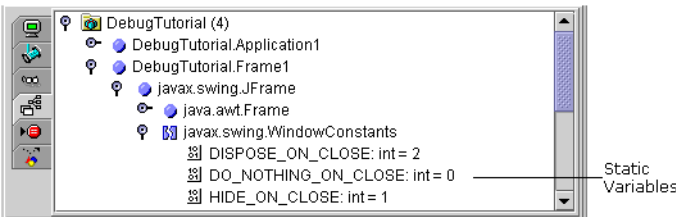
## How variables are displayed in the debugger

Variables can have different scope—there are static, or class variables, local variables, and member variables. A variable can hold a single value, such as a scalar (a single number), or multiple values, such as an array.

### Static variable display

Static variables are displayed in the Loaded classes and static data view. When you expand the tree of loaded classes, all static variables defined for a class and their values are displayed. You can use the context menu to set watches on these variables, change values of primitive data types, or toggle the base display value.

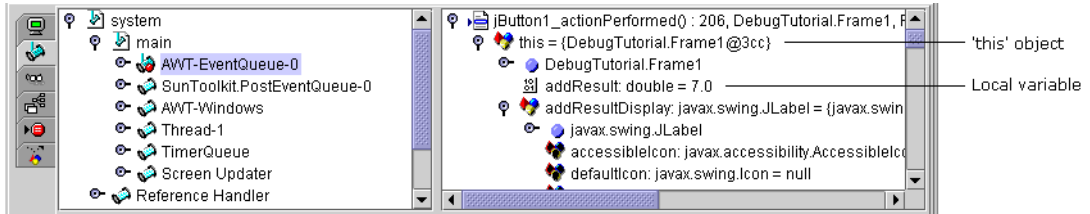
**Figure 8.14** Loaded classes and static data view



### Local and member variable display

Local and member variables are displayed in the Threads, call stacks, and data view. When you expand a thread group, a stack frame trace representing the current method call sequence is displayed. To display member variables, when you are in the class itself, expand the `this` object. When you expand that node, you see all of the variables that are members of that class and the instantiation that you are working through. Grayed-out elements are inherited.

You can then use the context menu to set watches on these variables, and, if the variable is an array, create an array watch and determine how many array elements will display in the view. You can also create a watch for the `this` object.

**Figure 8.15** Threads, call stacks, and data view

**Note** The split window is a feature of JBuilder SE and Enterprise.

## Changing data values

You can use the Data watches view, the Threads, call stacks, and data view, and the Loaded classes and static data view to examine and modify data values for variables.

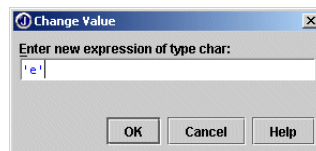
You can directly edit the value of a string or any primitive data type, including numbers and booleans, by right-clicking and choosing Change Value. (This is a feature of JBuilder SE and Enterprise.)

### Changing variable values

To change the value of a variable,

- 1 Select the variable whose value you want to modify.
- 2 Right-click and choose Change Value.

The Change Value dialog box is displayed.



- 3 Enter the new value. The new value must match the type of the existing value. The dialog box instructions state what type of value is expected. If the type doesn't match, the value will not be changed. A *String* constant must be surrounded by opening and closing " characters; a *char* constant value must be surrounded by opening and closing ' characters.
- 4 Click OK.

To change the display base of a numeric variable, right-click and choose Show Hex/Decimal Value. This command is a toggle—if the value is displayed in hex, it will display in decimal and vice versa.

This is a feature of  
JBuilder SE and  
Enterprise

**Note** In these dialog boxes, the previous selection is remembered and preserved. For example, when a node is redrawn, a breakpoint hit or a stepping button clicked, the value in the dialog box remains the same.

### Changing object/primitive variables

This is a feature of  
JBuilder SE and  
Enterprise


Object/primitive variables can be cut, copied, and pasted into other objects/primitives by using the Cut, Copy, and Paste commands on the context menus. If an object is pasted into another variable, both object variables will point to the same object. (The Cut, Copy, and Paste commands are features of JBuilder SE and Enterprise.)

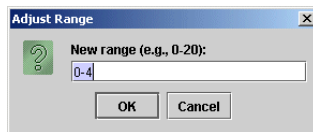
### Changing variable array values

You can change the value of an array element by right-clicking on the element you want to change and choosing Change Value. See [“Changing data values” on page 8-57](#) for more information.

You can also change how the array is displayed. You can view the details of the array by expanding it. However, there might be so many items displayed that you’ll have to scroll in the view to see all the values. For easier viewing, you can decrease or increase the number of items shown with the Adjust Range dialog box. By default, only the first 50 elements of an array are displayed.

To reduce the number of array elements displayed,

- 1 Right-click the array (the item in the view preceded by ) and choose Adjust Display Range.
- 2 The Adjust Range dialog box is displayed.



- 3 Enter the number of array elements you want to see.
- 4 Click OK.

You can also hide or display a null value in an array variable. This is useful when debugging a hash-map object. To enable this feature, right-click an array of type `Object` and choose Show/Hide Null Value. (This is a feature of JBuilder SE and Enterprise.)

**Note** In this dialog box, the previous selection is remembered and preserved. For example, when a node is redrawn, a breakpoint hit or a stepping button clicked, the value in the dialog box remains the same.

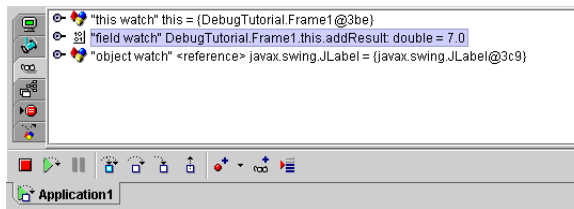


## Watching expressions

Watches enable you to monitor the changing values of variables or expressions during your program run. After you enter a watch expression, the Data watches view displays the current value of the expression. As you step through your program, watch expressions will be evaluated when they are in scope.

Watch expressions that return an object or array value can be expanded to show the data elements that are in scope. For example, if you set a watch on a `this` object, or on a single object, the watch can be expanded. However, if you set a watch on a primitive value, the watch can't be expanded since it's a single item. The grayed-out items in the expanded view are inherited.

**Figure 8.16** Data watches view



You can set two types of watches:

- Variable watches
- Object watches

### Variable watches

There are two types of variable watches:

- Named variable watches
- Scoped variable watches

#### Named variable watches

A named variable watch is added on a name—as you move around in your code, whatever variable has the name you selected in the current context will be the one evaluated for the watch. If no variable has the name you select, the debugger will display the following message in the Data watches view:

```
variable name = <is not in scope>
```

To add a named watch, use the Add Watch dialog box. To display this dialog box,

- Choose Run | Add Watch. Enter the expression to watch in the Expression field. You can optionally enter a description in the Description field.
- In the editor, select the expression you want to monitor. Then right-click, and choose Add Watch. The expression is automatically entered in the Add Watch dialog box. You can optionally enter a description in the Description field.

### Scoped variable watches

This is a feature of  
JBuilder SE and  
Enterprise

A scoped variable watch watches the variable in the scope (or context) in which you created the watch. As you move around in code, only the specific variable's value will be displayed. Scoped variable watches are features of JBuilder SE and Enterprise.

If the scoped variable watch expression is not in scope, the debugger will display the following message:

```
variable name = <is not in scope>
```







When the expression is in scope, the debugger will display its value.

To add a scoped variable watch, choose the variable you want to watch from the Threads, calls stacks, and data view. Then, right-click. The menu allows you to add a scoped watch for the selected type of variable, including:

- Field—A Java variable defined in a Java object.
- Static field—A Java variable defined as static (a class variable).
- Local variable—A variable that is local to a method or constructor.
- 'this' object—The class instantiation you are working through.
- Array—A collection of identical objects.
- Array component—An individual array element.
- String—A Java `String` type.

The following table shows how some of these types of watches are displayed in the Data watches view:

**Table 8.25** Types of scoped variable watches

Watch types	Display	Description
Field watch	 "addResult"DebugTutorial.Frame1.this.addResult: double=68.0	The field being watched, <code>addResult</code> , is a <b>primitive type</b> . It is in <code>DebugTutorial.Frame1</code> . Its value is 68.0.
Local variable watch	 "valueOneDouble"valueOneDouble: java.lang.double={java.lang.Double@354}	The local variable being watched is <code>valueOneDouble</code> . It is defined as a <b>Double object</b> .
Object watch	 "DebugTutorial.Frame1"<reference>DebugTutorial.Frame1= {DebugTutorial.Frame1@353}	The object being watched is <code>DebugTutorial.Frame1</code> . The object expands to show data members.
this watch	 "this" this:{DebugTutorial.Frame1@353}	The current instantiation of <code>DebugTutorial.Frame1</code> . The object expands to show data members for the current instantiation.
Array watch	 "valueOneText"<reference>char[]=char[2]	The array being watched is called <code>valueOneText</code> . It contains two array elements.
Array component watch	 "[0] = '3'"	The first element of the array <code>valueOneText</code> . It contains the value '3.'

## Object watches

This is a feature of  
JBuilder SE and  
Enterprise

An object watch watches a specific Java object.

To add an object watch,

- 1 Choose the object you want to watch. You can choose an object in the Data watches view, the Threads, calls stacks, and data view, or the Loaded classes and static data view.
- 2 Right-click and choose Create Object Watch.

A this object watch watches the current instantiation of the selected object.

## Editing a watch

To edit a watch expression, select the expression in the Data watches view, then right-click. Choose Change Watch.

- To change the watch name, enter the new name in the Expression field.
- To change the watch description, enter the new name in the Description field.

## Deleting a watch

To delete a watch expression, select the expression in the Data watches view, then select Remove Watch or press *Delete*. You can delete all watches by right-clicking in an empty area of the Data watches view and choosing Remove All.

**Caution** The Remove All command cannot be reversed.

## Evaluating and modifying expressions

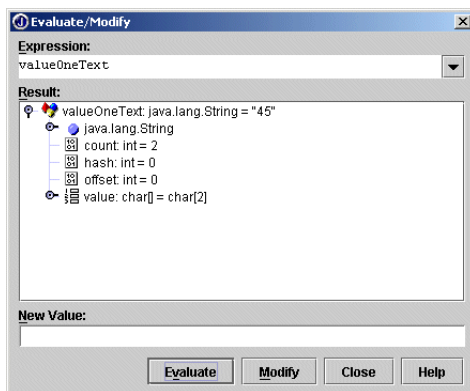
You can evaluate expressions, change the values of data items, and evaluate method calls with the Evaluate/Modify dialog box (Run | Evaluate/Modify). This can be useful if you think you've found the solution to a bug, and you want to try the correction before exiting the debugger, changing the source code, and recompiling the program. CodeInsight and syntax highlighting display when you enter an expression into the Expression field.

To open the Evaluate/Modify dialog box, choose Run | Evaluate/Modify.

### Evaluating expressions

To evaluate an expression, enter the expression in the Expression field. If the expression is already selected in the editor, it is automatically entered into the Expression field. Then, click the Evaluate button. You can use this dialog box to evaluate any valid language expression, except expressions that are outside the current scope. If the result is an object, note that the contents of the object are displayed.

**Figure 8.17** Expression evaluation in the Evaluate/Modify dialog box



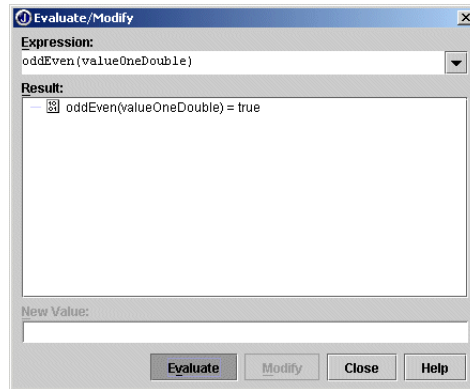
### Evaluating method calls

This is a feature of  
JBuilder SE and  
Enterprise

The results of a method call can also be evaluated. To evaluate a method call, enter the method and its parameters in the Expression field of the Evaluate/Modify dialog box. Click Evaluate.

In this example, the method return value evaluated to `true`.

**Figure 8.18** Method evaluation in the Evaluate/Modify dialog box



## Modifying the values of variables

This is a feature of  
JBuilder SE and  
Enterprise

You can change the values of variables during the course of a debugging session to test different error hypotheses and see how a section of code behaves under different circumstances.

When you modify the value of a variable through the debugger, the modification is effective for that specific program run only—the changes you make through the Evaluate/Modify dialog box do not affect your program source code or the compiled program. To make your change permanent, you must modify your program source code in the editor, then recompile your program.

To modify the value of a variable, enter:

```
variable = <new value>
```

in the Expression edit box. The debugger will display the results in the Result display box. Note that the result must evaluate to a result that is type compatible with the variable.

**Note** Both the Expression and New Value fields support CodeInsight.

You can also modify the value of a variable using these steps:

- 1 Open the Evaluate/Modify dialog box, then enter the name of the variable you want to modify in the Expression edit box.
- 2 Click Evaluate to evaluate the variable.
- 3 Enter a value into the New Value edit box (or select a value from the drop-down list), then click Modify to update the variable.

The expression in the Expression input box or the new value in the New Value box needs to evaluate to a result that is type-compatible with the variable you want to assign it to. In general, if the assignment would cause a compile-time or runtime error, it's not a legal modification value.

For example, assume that `valueOneText` is a `String` object. If you enter:

```
valueOneText=34
```

in the Expression input field, the following message, indicating a type mismatch, would be displayed in the Results field:

```
incompatible types; found int; required java.lang.String
```

You would need to enter:

```
valueOneDouble="34"
```

in the Expression input field in order for the expression to be set to the new value.

## Modifying code while debugging

---

This is a feature of  
JBuilder Enterprise

The debugger allows you to make changes in source code while debugging. You can either update all class files in your project, or update individual ones.

### Updating all class files

---



When files are modified while debugging, the Smart Swap button on the debugger toolbar is available. When you click this button, all modified files in your project are compiled and updated. With Smart Swap, you can test your code, make changes, and continue debugging in the same debugger session, from the current execution point.

To use Smart Swap,

- 1 Open the source code for the file(s) you want to modify.
- 2 Change the source code.
- 3 Click the Smart Swap button or choose Run | Smart Swap.



Smart Swap compiles all modified files in the project. You will continue debugging in the same debugging session.

#### Important

If the Target VM for your project (Project Properties | Build | Java) is set to All Java SDKs, Smart Swap may not work properly. The All Java SDKs Target VM option generates class files that can be loaded by any VM. This class file, however, will include code that instructs the class loader to verify all classes that are referenced in this class. If the referenced classes have not been loaded yet, the class loader will load the class file in its cache. Smart Swap cannot access this cache to update it. Consequently, when you are debugging and make a change to a class file that has not yet been loaded, the debugger will not be able to see the changes that you have made. To work around this, add the following VM parameter to the VM Parameters field on the Run page of the Runtime Configurations

dialog box. (You only need to add this parameter when selecting All Java SDKs for the target VM.)

```
-Xverify:none
```

This VM parameter instructs the class loader to verify only the current class.

## Updating individual class files

---

While debugging, you can also update just individual classes in your project. To update a single class file while debugging,

- 1 Before debugging, make sure the Update Classes After Compiling option on the Debug page of the Runtime Configuration Properties dialog box is on.
- 2 While debugging, open the source code for the file(s) you want to modify.
- 3 Right-click the file in the project pane and choose Make. JBuilder will automatically compile the file and update the classes. You will continue in the same debugging session.

**Note** If the Update Classes After Compiling option is off (Debug page | Runtime Configuration Properties dialog box) and the Warn If Files Modified option on that page is on, the Files Modified dialog box is displayed, where you choose how to proceed. You can:

- Compile, update the compiled classes, and continue the debugging session
- Resume the debugging session without compiling and updating
- Restart the debugging session

## Resetting the execution point

---

Once you've modified code, you might want to reset the execution point (you'll still be in the same debugging session, however). Resetting the execution point allows you to return to a point prior to your changed value, so you can retest it to see if your fix works. For more information, see ["Setting the execution point" on page 8-29](#).

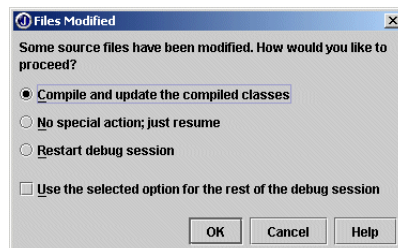
## Options for modifying code

---

The options at the top of the Debug page of the Runtime Configuration Properties dialog box control how files modified during a debugging session are handled.

**Figure 8.19** Debug page of Runtime Configuration Properties dialog box

- The Update Classes After Compiling option automatically updates any changed file(s) that are compiled. When this option is on, you won't be warned that you've modified source code if you compile the modified files. If this option is off and the Warn If Files Modified option is on, the Files Modified dialog box will be displayed, allowing you to determine how to continue.
- The Warn If Files Modified option displays the Files Modified dialog box, where you choose how to continue. The Files Modified dialog box looks like this:





You can choose to:

- Update files and continue in the same debugging session,
- Resume debugging without updating files, or
- Start a new debugging session.

If you start a new session, the current session will be stopped and the execution point will be reset to the beginning of the program. If you resume the current session, you will begin execution at the current execution point.

## Customizing the debugger

You can customize the colors used to indicate the execution point and enabled, disabled, and invalid breakpoint lines.

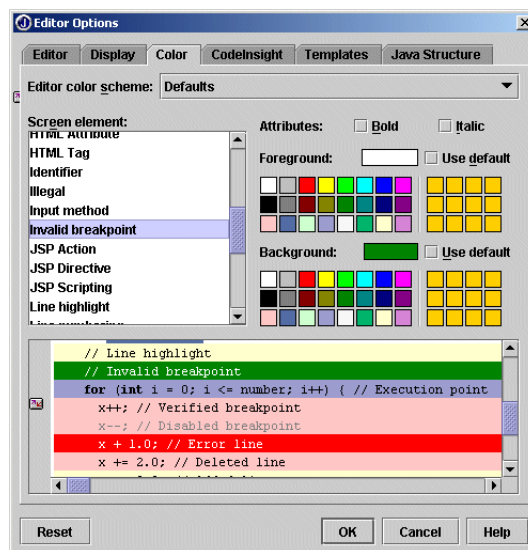
### Customizing the debugger display

To set execution point and breakpoint colors,

- 1 Select Tools | Editor Options.

The Editor Options dialog box is displayed.

- 2 Select the Color tab to display the Color page.



- 3 In the Screen Element list, select an element related to debugging, then select the background and foreground colors for the element.

Screen elements related to debugging are:

- Default breakpoint
- Disabled breakpoint
- Execution backtrace
- Execution point
- Invalid breakpoint
- Verified breakpoint

## Setting debug configuration options

Multiple debug configurations are a feature of JBuilder SE and Enterprise

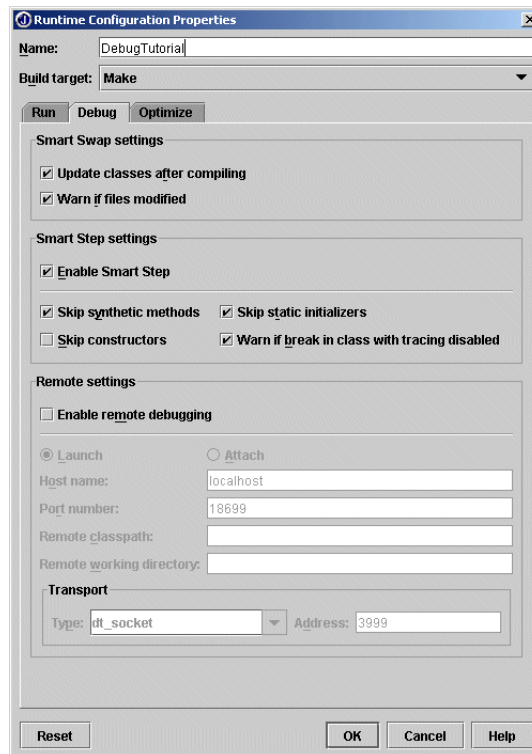
You can either create a stand-alone debug configuration or create one as part of a runtime configuration. For information on runtime configurations, see [“Setting runtime configurations” on page 7-6](#).

To set debug configuration options,

- 1 Choose Run | Configurations.

The Run page of the Project Properties dialog box is displayed.

- 2 Choose the configuration you want to edit and click Edit.
- 3 In the Runtime Configuration Properties dialog box, select the Debug tab.



- 4 To configure how the debugger handles modified files, set the following options.

- Update Classes After Compiling

Automatically updates any changed file(s) when compiled. You won't be warned that you've modified source code if this source code has been compiled. If this option is off and the Warn If Files Modified option is on, the Files Modified dialog box will be displayed, allowing you to determine how to continue. (This is a feature of JBuilder Enterprise.)

- Warn If Files Modified

Displays the Files Modified dialog box, where you choose how to continue. You can update files and continue the current debugging session, resume debugging without updating files, or start a new debugging session.



- 5 To enable Smart Step, choose the Enable Smart Step option or click the Smart Step button on the debugger toolbar. Smart Step is on by default.

- 6 To configure the Smart Step toggle, set the following options. (Smart Step configuration is a feature of JBuilder SE and Enterprise.)

- Skip synthetic methods

Skips synthetic methods when stepping into classes.

- Skip constructors

Skips constructors when stepping into classes.

- Skip static initializers

Skips static (class) initializers when stepping into classes.

- Warn if break in class with tracing disabled

Displays a warning message if there is a breakpoint in a class that has tracing disabled. See [“Breakpoints and tracing disabled settings” on page 8-40](#) for more information.

For information on remote debugging options, see [Chapter 9, “Remote debugging.”](#) (Remote debugging is a feature of JBuilder Enterprise.)

## Setting update intervals

---

You can also specify the frequency of the intervals that control when console/process state changes are polled. If the intervals are small, the debugger/runtime responses for output and other events, like stepping, will be faster, but JBuilder will be using most of the CPU time.

In general, you can make these settings small, unless you are running other applications along with JBuilder, or the program you are debugging

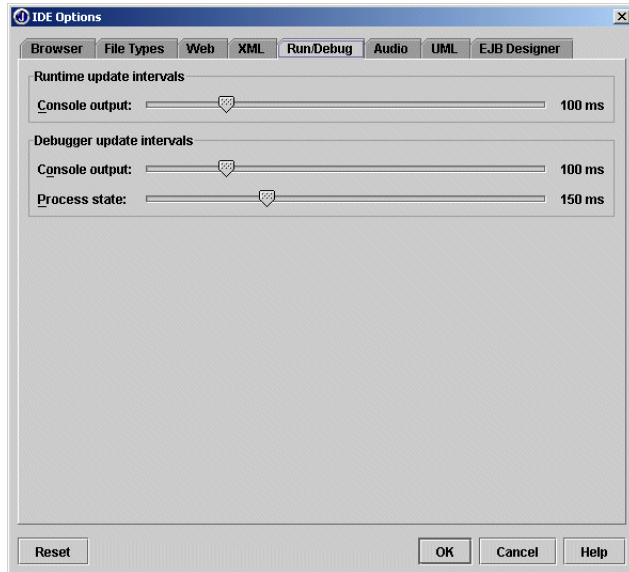
requires a lot of CPU time. If this is the case, you should make the intervals larger.

To change the update intervals,

- 1 Choose Tools | IDE Options.

The IDE Options dialog box is displayed.

- 2 Select the Run/Debug tab to display the Run/Debug page.



- 3 To set the interval for console output for runtime processes, move the Console Output slider bar at the top of the dialog box.
- 4 To set the interval for console output for debug processes, move the Console Output slider bar in the middle of the dialog box.
- 5 To set the interval for debug process state updates, move the Process State slider bar.

## Remote debugging

This is a feature of  
JBuilder Enterprise

JBuilder includes several debugger features that assist in debugging distributed applications. In particular, it includes support for cross-process debugging and remote debugging.

This support is additional to the basic debugging features in JBuilder. If you are new to JBuilder, refer to [Chapter 8, “Debugging Java programs”](#) for information on the JBuilder debugger environment.

Remote debugging is the process of debugging code running on one computer from another computer. This feature is ideal, for example, in situations where an application encounters a problem on one networked computer that is not duplicated on other computers. With JBuilder’s remote debugger, you can also debug across operating system platforms.

In this chapter, the “client computer” is the computer running JBuilder. This is the computer you debug from. The “remote computer” runs the application you want to debug.

There are two ways to debug remotely. You can either

- Launch a program on the remote computer from the client computer and debug it using JBuilder on the client computer. For more information, see [“Launching and debugging a program on a remote computer” on page 9-2](#). In this case, you run JBuilder’s Debug Server on the remote computer.
- Attach to a program already running on the remote computer and debug it using JBuilder on the client computer. For more information, see [“Debugging a program already running on the remote computer” on page 9-6](#). In this case, you don’t need to run the Debug Server.

**Note** Both the client and remote computer must have JDK 1.2 or higher installed (the JDK must support the JPDA debugging API). The JDK versions on the

two computers do not need to match. Note that when you install JBuilder, JDK 1.4 is automatically installed in the `<jbuilder>/jdk1.4` directory.

You can also debug local code that is running in a separate process on the same computer JBuilder is installed on. To do this, start the process in debug mode and attach JBuilder to it. For more information, see [“Debugging local code running in a separate process” on page 9-9](#).

Additionally, you can set cross-process breakpoints, which is ideal for debugging client/server applications. For more information, see [“Debugging with cross-process breakpoints” on page 9-10](#).

For any type of remote debugging, you need to debug through a debug configuration. For more information on run and debug configurations, see [“Setting runtime configurations” on page 7-6](#) and [“Setting debug configuration options” on page 8-68](#).

## Launching and debugging a program on a remote computer

---

This section explains how to launch a program on a remote computer and debug it using JBuilder on the client computer. Briefly, you

- 1 Install the Debug Server on the remote computer and run it.
- 2 Either compile the application on the remote computer or copy the application's `.class` files to the remote computer.
- 3 Use JBuilder on the client computer to launch and debug the application on the remote computer.

**Important** The source files for the application you are debugging must be available on the client computer. The compiled `.class` files must be available on the remote computer. They must match. Otherwise, unpredictable results may occur, including incorrect errors being generated or the debugger stopping on the wrong source code line. Every time you modify the source code, be sure to update `.class` files on the remote computer.

First, install and run the Debug Server on the remote computer. If JBuilder is already installed on the remote computer, you can start with Step 4 below.

- 1 Copy the `debugserver.jar` file (located in the `<jbuilder>/remote` directory) to the remote computer. Note the directory location you copy it to as it will be needed in later steps.
- 2 Copy the Debug Server shell script, `DebugServer` (Unix), or batch file, `DebugServer.bat` (Windows), to the same directory on the remote computer.
- 3 Make sure that JDK 1.2.2 or higher is installed on the remote computer.

- 4 Go to the directory on the remote computer where the Debug Server files are installed. Run `DebugServer` to customize environment variables for the remote Debug Server.

For Unix systems, use the following command:

```
./DebugServer <debugserver.jar_dir> <jdk_home_dir> [-port portnumber]
[-timeout milliseconds]
```

For Windows systems, use:

```
DebugServer <debugserver.jar_dir> <jdk_home_dir> [-port portnumber]
[-timeout milliseconds]
```

where:

- `debugserver.jar_dir` - The directory on the remote computer where the Debug Server JAR file is located. On Windows systems, the drive letter is required.
- `jdk_home_dir` - The home directory on the remote computer of the JDK installation. On Windows systems, the drive letter is required.
- `-port` - Optional parameter that launches the debug server on a port different from the default, 18699. Change this value only if the default value is in use. Valid values are from 1025 to 65535. This value must match the value entered into the Port Number field on the Debug page of the Runtime Configuration Properties dialog box (on the client computer). See Step 6 below.
- `-timeout` - Optional parameter that sets the number of milliseconds to try to connect the remote computer to the client computer. When this number is reached, the process will stop. The default setting is 60,000 milliseconds.

An example of this command in a Windows environment is:

```
DebugServer d:\remote d:\jdk1.3 -port 1234 -timeout 20000
```

- 5 Press *Enter* to start the Debug Server.

When the Debug Server is loaded, JBuilder remote debugging functionality in launch mode is enabled. Once the Debug Server is running, you need to compile the application and copy the `.class` files to the remote computer. (You can also compile the application remotely.) Then, you use JBuilder, running on the client computer, to launch and debug the program on the remote computer.

- 1 Compile the application. You can compile the application using JBuilder on the client computer, then copy or use File Transfer Protocol (FTP) to put `.class` files on the remote computer. You can also compile the application directly on the remote computer using the `-g` option when calling the `javac` compiler. (This tells the compiler to add debug information to the compiled file.)
- 2 Open JBuilder on the client computer.

- 3 Open the project for the application to debug.
- 4 Create a debug configuration (it can be part of an existing runtime configuration or a new configuration):
  - a Choose Run | Configurations. To create a new configuration, click the New button. To edit a configuration, choose it and click Edit.
  - b For a new configuration, enter a name in the Configuration Name field.
  - c Set the Build Target to None.
  - d Select the Debug tab.



- 5 Select the Enable Remote Debugging check box. Select the Launch option.
- 6 Fill in the following fields:
  - Host Name - The name of the remote computer. localhost is the default value. You may need to check the network settings on the remote computer to find the host name.
  - Port Number - The port number for the remote computer you are communicating with. Use the default port number, 18699. Change this value only if the default value is in use. Valid values are from



1024 to 65535. This value must match the `-port` parameter used to launch the Debug Server on the remote computer. See Step 4 in the previous section.

- **Remote Classpath** - The classpath where the compiled `.class` files for the application that you are remotely debugging can be found. This field works like other classpath fields - if the classes are in a package, specify the root of the package and not the directory containing the classes. On Windows systems, specify the drive letter if other than `C:`. This remote classpath only applies to this debugging session.
- **Remote Working Directory** - The working directory on the remote computer. On Windows systems, specify the drive letter if other than `C:`. This remote working directory only applies to this debugging session.

**Warning**

The working directory is not supported in JDK 1.2.2. If your remote computer is running JDK 1.2.2 and you enter a remote working directory, the debugger will display a warning in the Console output, input, and errors view.

- **Transport** - The transport type: Either `dt_shmem` (shared memory transport - not available on Unix systems) or `dt_socket` (socket transport). For more information on transport methods, see “JPDA: Connection and Invocation Details - Transports” at <http://java.sun.com/products/jpda/doc/conninv.html#Transports>.
- 7 Click OK two times to close the Runtime Configuration Properties and Project Properties dialog boxes.
  - 8 Start the debugging session by choosing one of the following options:

Command	Shortcut	Description
Run   Debug Project	<i>Shift + F9</i>	Starts the program in the debugger using the default or selected configuration. Either runs the program to completion or suspends execution at the first line of code where user input is required, or at a breakpoint.
Run   Step Over	<i>F8</i>	Suspends execution at the first line of executable code.
Run   Step Into	<i>F7</i>	Suspends execution at the first line of executable code.

Once you start the debugger, the application you want to debug (based on the Remote Classpath setting) is launched on the remote computer. The debugger is displayed in JBuilder running on the client computer; however, you are debugging the `.class` files running on the remote computer.

**Note**

If the application is already running on the remote computer, the Debug Server will launch a new instance of it. (To debug an

already-running application, see [“Debugging a program already running on the remote computer” on page 9-6.](#))

- 9 To terminate the application on the remote computer, stop the process in JBuilder. To close the Debug Server on the remote computer, choose the Debug Server’s File | Exit command.

## Debugging a program already running on the remote computer

---

This section explains how to attach to a program that is already running on a remote computer and debug it using JBuilder on the client computer. To do so, you will:

- 1 Run the application with VM debug options on the remote computer.
- 2 Use JBuilder on the client computer to attach to and debug the running application.

**Important** The source files for the application you are debugging must be available on the client computer. The compiled `.class` files must be available on the remote computer. They must match. Otherwise, unpredictable results may occur, including incorrect errors being generated or the debugger stopping on the wrong source code line. Every time you modify the source code, be sure to update the `.class` files on the remote computer.

For a tutorial that walks through attaching to an already running program, see [Chapter 19, “Tutorial: Remote debugging.”](#)

To start a program on the remote computer and attach to it,

- 1 Compile the application on the remote computer. You can also compile the application in JBuilder on the client computer, then copy or use File Transfer Protocol (FTP) to put `.class` files on the remote computer.
- 2 Run the application on the remote computer, using the following VM options.
  - If JBuilder is installed on the remote computer, you can run your program from within JBuilder. Open the project, then edit the runtime configuration (Run | Configurations | Edit). Enter the following parameters into the VM Parameters field:

```
-Xdebug -Xnoagent -Djava.compiler=NONE  
-Xrunjdwp:transport=dt_socket,server=y,address=3999,suspend=y
```

- If you don't have JBuilder on the remote computer, you need to run your program from the command line. Add the following VM options to the Java command line:

```
-Xdebug -Xnoagent -Djava.compiler=NONE  
-Xrunjdpw:transport=dt_socket,server=y,address=3999,suspend=y
```

The `address` and `suspend` parameters are optional. They follow the `server` parameter and are separated by a comma. No spaces are allowed between the parameters.

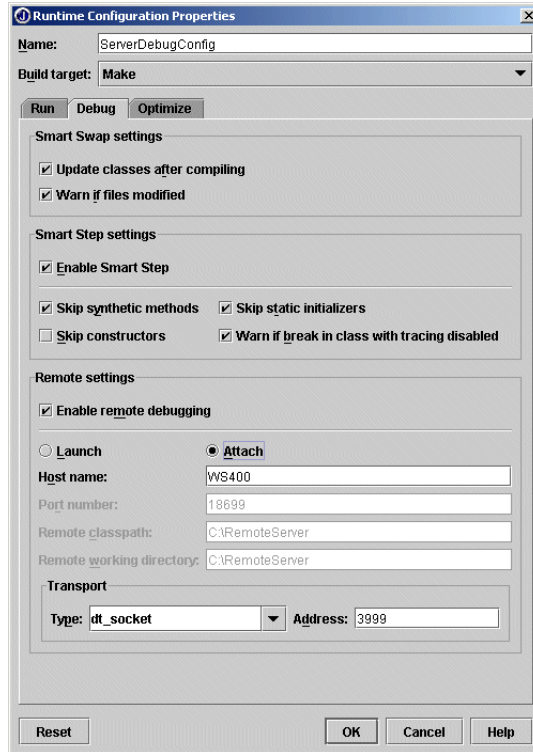
- The `address` parameter, based on the selected transport, holds the port number/address through which the debugger communicates with the remote computer. This parameter makes configuration easier - you don't need to continually modify the Address field on the Debug page of the Runtime Properties dialog box. If the Transport Type is set to `dt_socket`, the `address` parameter holds the port number. If it's set to `dt_shmem`, this parameter is set to the unique address name. If you're using XP, do not use 5000 for the `address` parameter. This address is reserved for the Universal Plug & Play.
- The `suspend` parameter indicates whether the program is suspended immediately when it is started. You can turn off this setting by specifying `suspend=n`. (If `suspend=n` and no breakpoints are set, the program will run to completion without stopping when you start it.)

**Note**

To run the application with JDK 1.2x or 1.3x, use the `java` executable from the `bin` directory of your JDK installation, not the `java.exe` in the `jre/bin` directory. This allows the Java VM to load the debugger file (`libjdpw.so` in Unix; `jdpw.dll` in Windows), which is necessary for debugging. (This does not apply to JDK 1.4x.)

- 3 Open JBuilder on the client computer.
- 4 Open the project for the application already running on the remote computer.
- 5 Create a debug configuration (it can be part of an existing runtime configuration or a new configuration):
  - a Choose Run | Configurations. To create a new configuration, click the New button. To edit a configuration, choose it and click Edit.
  - b For a new configuration, enter a name in the Configuration Name field.
  - c Set the Build Target to None.

d Select the Debug tab.



6 Select the Enable Remote Debugging checkbox. Select the Attach option.

7 Fill in the following fields:

- Host Name - The name of the remote computer. localhost is the default. You may need to check the network settings on the remote computer to find the host name.
- Transport - The transport method options:
  - Type: Either dt\_socket (socket transport) or dt\_shmem (shared memory transport - not available on Unix systems). For more information on transport methods, see "JPDA: Connection and Invocation Details - Transports." at <http://java.sun.com/products/jpda/doc/conninv.html#Transports>.
  - Address:
    - If the Transport Type is set to dt\_socket, this parameter holds the port number for the remote computer you are communicating with. Use the default port number, 3999. Change this value only if the default value is in use. This value must match the address

**Important**

parameter to the Java VM that starts the program on the remote computer. See Step 2 earlier in this section.

If you are running Windows XP, do not use 5000 as the port number/address through which the debugger communicates with a remote computer. XP reserves this port for the Universal Plug & Play.

- If `dt_shmem` is selected as the Transport Type, set the `address` parameter to the unique name for the remote computer you are communicating with. The default is `javadebug`.

8 Click OK two times to close the Runtime Configuration Properties and Project Properties dialog boxes.

9 Choose either Run | Step Over or Run | Step Into to start the debugger.



10 If the `suspend` parameter for the VM on the remote computer is set to `y` (see Step 2 earlier in this section), click the Resume Program button on the debugger toolbar to proceed with debugging.

11 To terminate the application, close the application on the remote computer.

12 To detach from the remote computer, stop the process in JBuilder.

**Note** To launch and debug an application on the remote computer, see [“Launching and debugging a program on a remote computer” on page 9-2.](#)

## Debugging local code running in a separate process

---

To debug local code that is running in a separate process on the same computer JBuilder is installed on, follow the instructions above, starting with Step 2. Use the following settings for the Attach options on the Debug page of the Runtime Configuration Properties dialog box:

- Host Name  
Set to the default, `localhost`.
- Transport Type  
Set to `dt_socket` (socket transport) or `dt_shmem` (shared memory transport - not available on Unix systems).
- Transport Address  
If the Transport Type is `dt_socket`, set to 3999. If the Transport Type is `dt_shmem`, set to `javadebug`.

## Debugging with cross-process breakpoints

---

A cross-process breakpoint causes the debugger to stop when you step into any method or the specified method in the specified class in a separate process. This allows you to step into a server process from a client process, rather than having to set breakpoints on the client side and on the server side. You will usually set a line breakpoint on the client side and a cross-process breakpoint on the server side. For a tutorial that demonstrates cross-process stepping, see [Chapter 19, “Tutorial: Remote debugging.”](#)

To activate a cross-process breakpoint set on a server process,

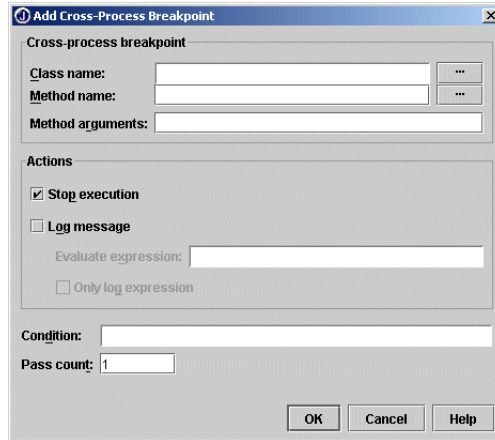
- 1 Start the server process on the remote computer in debug mode. See Step 2 in [“Debugging a program already running on the remote computer”](#) on page 9-6.
- 2 On the client computer, from within JBuilder, attach to the server already running on the remote computer. See Steps 4 - 10 in [“Debugging a program already running on the remote computer”](#) on page 9-6.
- 3 Set a line breakpoint in the client code and start debugging the client. At the breakpoint, step into the server code. Do not use Step Over. Stepping over will not stop at the cross-process breakpoint.

To set a cross-process breakpoint, use the Add Cross-Process Breakpoint dialog box. To open the Add Cross-Process Breakpoint dialog box, do one of the following:

- Before or during a debugging session, select Run | Add Breakpoint and choose Add Cross Process Breakpoint.
- When you're in a debugging session, click the down-facing arrow to the right of the Add Breakpoint button on the debugger toolbar and choose Add Cross-Process Breakpoint.
- When you're in a debugging session, right-click an empty area of the Data and code breakpoints view and choose Add Cross-Process Breakpoint.



The Add Cross-Process Breakpoint dialog box is displayed.



To set a cross-process breakpoint,

- 1 In the Class Name field, enter the name of the server-side class that contains the method you want the debugger to stop on. Use the Browse button to browse to the class.
- 2 In the Method field, enter the name of the method you want the debugger to stop on. Use the Browse button to display the Select Method dialog box where you can browse through the methods available in the selected class. The method name is not required. If you do not specify the method name, the debugger stops at all method calls in the specified class.

**Note** You cannot select a method if the selected class contains syntax or compiler errors.

- 3 In the Method Arguments field, enter a comma-delimited list of method arguments. The debugger will stop when the method name and argument list match. This is useful for overloaded methods.
  - If you don't specify any arguments, the debugger stops at all methods with the specified method name.
  - If you select a method name from the Select Method dialog box, the Methods Argument field is automatically filled in.
- 4 Choose the Actions for the debugger. The debugger can stop execution at the breakpoint, display a message, or evaluate an expression. For more information, see [“Setting breakpoint actions” on page 8-51](#).
- 5 In the Condition field, set the condition, if one exists, for this breakpoint. For more information, see [“Creating conditional breakpoints” on page 8-52](#).

- 6 In the Pass Count field, set the number of times this breakpoint must be passed in order for the breakpoint to be activated. For more information, see [“Using pass count breakpoints” on page 8-53](#).
- 7 Click OK to close the dialog box.
- 8 Set a line breakpoint in the client on the method that calls the cross-process breakpoint.
- 9 When you stop at the line breakpoint, click the Step Into button on the debugger toolbar to step into the server-side breakpointed method. (If you use Step Over, the debugger will not stop.)





# Chapter 10

## Creating JavaBeans with BeansExpress

This is a feature of  
JBuilder SE and  
Enterprise

BeansExpress is the fastest way to create JavaBeans. It consists of a set of wizards, visual designers, and code samples that help you build JavaBeans rapidly and easily. Once you have a JavaBean, you can use BeansExpress to make changes to it. Or you can take an existing Java class and turn it into a JavaBean.

### What is a JavaBean?

---

A JavaBean is a collection of one or more Java classes, often bundled into a single JAR (Java Archive) file, that serves as a self-contained, reusable component. A JavaBean can be a discrete component used in building a user interface, or a non-UI component such as a data module or computation engine.

At its simplest, a JavaBean is a `public` Java class that has a constructor with no parameters. JavaBeans usually have properties, methods, and events that follow certain naming conventions (also known as design patterns).

### Why build JavaBeans?

---

Like other types of components, JavaBeans are reusable pieces of code that can be updated with minimal impact on the testing of the program they

become a part of. JavaBeans have some unique advantages over other components, however:

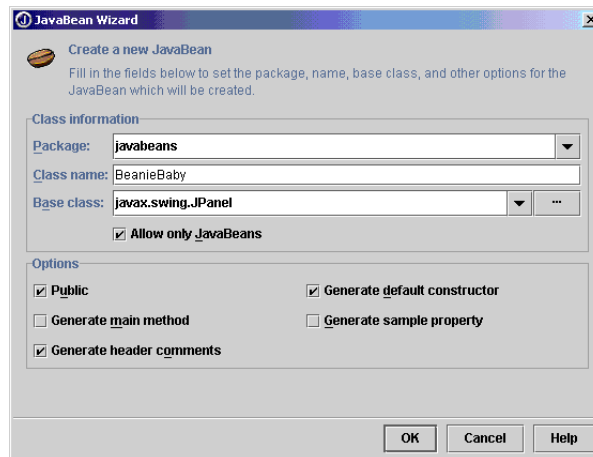
- They are pure Java, cross-platform components.
- You can install them on the JBuilder component palette and use them in the construction of your program, or they can be used in other application builder tools for Java.
- They can be deployed in JAR files.

## Generating a bean class

---

To begin creating a JavaBean using BeansExpress,

- 1 Choose File | New Project and create a new project with the Project wizard.
- 2 Choose File | New to display the object gallery.
- 3 Click the General tab and double-click the JavaBean icon to open the JavaBean wizard.



- 4 Specify the package you want the bean to be part of in the first text field. By default, this will be the name of your current project.
- 5 Give your bean a name in the second text field.
- 6 Choose a class to extend in the Base Class field.

Either use the drop-down list, or click the adjacent button to display the Package browser and use it to specify any existing Java class you want.
- 7 Choose the remaining options you want; none of them are required:
  - a Check Allow Only JavaBeans if you want JBuilder to warn you if you try to extend a Java class that is not a valid JavaBean.

- b** Check **Public** if you want class to be `public`.
  - c** Check **Generate Main Method** if you want JBuilder to place the `main()` method within your bean that makes it runnable.
  - d** Check **Generate Header Comments** if you want JavaDoc header comments (Title, Description, Author, and so on) added to the top of your class file.
  - e** Check **Generate Default Constructor** if you want JBuilder to create a parameterless constructor.
  - f** Check **Generate Sample Property** if you want JBuilder to add a property called `sample` to your bean. You might want this for your first bean to see how JBuilder generates the required code for a property. You can remove this property later, or make it an actual property that your bean can use.
- 8** Choose **OK** to close the JavaBean wizard.

JBuilder creates a JavaBean with the name you specified, places it in your current project, and displays the source code it generated. This is the code JBuilder generates for the settings shown:

```
package myjavabean;

import java.awt.*;
import javax.swing.*;

public class BeanieBaby extends JPanel {
    BorderLayout borderLayout1 = new BorderLayout();

    public BeanieBaby() {
        try {
            jbInit();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setLayout(borderLayout1);
    }
}
```

If you examine the code JBuilder generated, you'll see that

- JBuilder named the bean as you specified and extended the designated class; the class is declared as `public`.
- The class has a parameterless constructor.

Even in this rudimentary state, your class is a valid JavaBean.

## Designing the user interface of your bean

---

Not all JavaBeans have a user interface, but if yours does, you can use JBuilder's UI designer to create it.

To create a user interface for your bean,

- 1 Select the bean file JBuilder created for you in the project pane.
- 2 Click the Design tab to display the UI designer.
- 3 Use the UI designer to build the user interface for your bean.

For information on creating a user interface, see *Designing Applications with JBuilder*.

## Adding properties to your bean

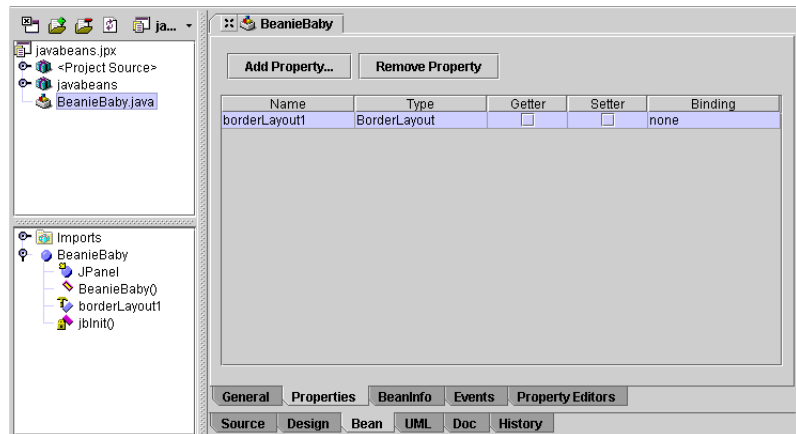
---

Properties define the attributes your bean has. For example, the `backgroundColor` property describes the color of the background.

JavaBeans properties usually have both a read and a write access method, also known as a getter and a setter, respectively. A getter method returns the current value of the property; a setter method sets the property to a new value.

To add a property to your bean,

- 1 Select your component in the project pane and click the Bean tab to display the BeansExpress designers.
- 2 Click the Properties tab to display the Properties designer.

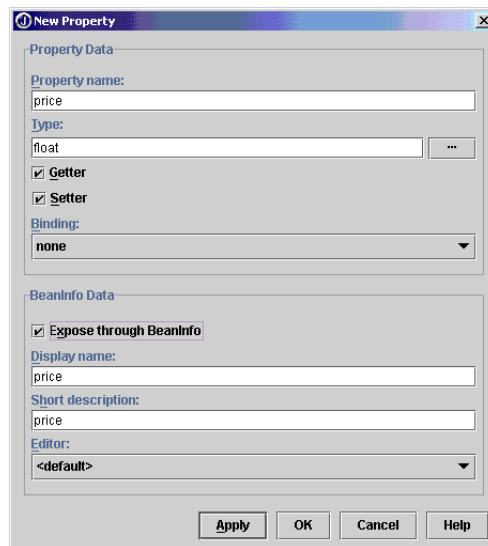


- 3 Choose the Add Property button. The New Property dialog box appears.

- 4 Specify the name of the property in the Property Name box.
- 5 Specify a type in the Type box.  
You can type in one of the Java types or use the Package browser to select any object type, including other JavaBeans.
- 6 Leave both Getter and Setter check boxes checked if you want JBuilder to generate the methods to both get and set the property value.  
If you want to make a property read-only, uncheck the Setter check box.
- 7 Choose OK.

JBuilder generates the necessary code for the property and adds the property to the Properties designer grid. You can see the read and write access methods added to the component tree for your bean. If you click the Source tab, you can see the code JBuilder generated.

Here is the New Property dialog box with all the required fields filled in:



The Display Name and Short Description are filled in automatically with default values. This is the resulting source code:

```
package myjavabean;

import java.awt.*;
import javax.swing.JPanel;

public class BeanieBaby extends JPanel {
    BorderLayout borderLayout1 = new BorderLayout();
    private float price;           // Added a private price field
}
```

## Adding properties to your bean

```
public BeanieBaby() {
    try {
        jbInit();
    }
    catch(Exception ex) {
        ex.printStackTrace();
    }
}

private void jbInit() throws Exception {
    this.setLayout(borderLayout1);
}

public void setPrice(float price) { // Added a method to change the price
    // value
    this.price = price;;
}

public float getPrice() { // Added a method to obtain the price
    // value
    return price;
}
}
```

JBuilder added a `private price` field to the `BeanieBaby` class. The `price` field holds the value of the property. Usually a property field should be declared as `private` so that only the getter and setter methods can access the field.

It also generated a `setPrice()` method that can change the value of the new field, and it generated a `getPrice()` method that can get the current value of the field.

When your bean is installed on JBuilder's component palette and users drop the bean on the UI designer, the price property will appear in the Inspector so that users can modify its value. All the code to get and set the price property is in place.

## Modifying a property

---

Once you've added a property to your bean, you can modify it at any time with the Properties designer.

To modify a property,

- 1 Select your bean using the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.

- 3 Click the Properties tab.
- 4 Select any of the fields in the Properties designer grid and make the changes you want.

For example, you can change the type of the property by entering another type.

JBuilder reflects the changes you make in the Properties designer in the source code of your bean. You can also make changes directly in the source code and the BeansExpress designers will reflect the changes if the changes have been made correctly.

## Removing a property

---

To remove a property from your bean,

- 1 Select the bean that contains the property in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the Properties tab.
- 4 Select the property you want removed in the Properties designer grid.
- 5 Click Remove Property.

The property field and its access methods are removed from the source code.

## Adding bound and constrained properties

---

BeansExpress can generate the necessary code to create bound and constrained properties.

To add a bound or constrained property,

- 1 From the Properties designer, click the Add Property button to display the New Property dialog box.
- 2 Specify a property name and type.
- 3 Use the Binding drop-down list to specify the property as bound or constrained.
- 4 Choose OK.

If the property you added is a bound property, JBuilder adds the `private` property field to the class and generates its read and write methods. It also instantiates an instance of the `PropertyChangeSupport` class. For example, here is the code added for a bound property called `allTiedUp`:

```
public class BeanieBaby extends JPanel {
    ...
    private String allTiedUp;
    private transient PropertyChangeSupport propertyChangeListeners = new
        PropertyChangeSupport(this);
    ...
    public void setAllTiedUp(String allTiedUp) {
        String oldAllTiedUp = this.allTiedUp;
        this.allTiedUp = allTiedUp;
        propertyChangeListeners.firePropertyChange("allTiedUp", oldAllTiedUp,
            allTiedUp);
    }
    public String getAllTiedUp() {
        return allTiedUp;
    }
}
```

Note that the `setAllTiedUp()` method includes the code to notify all listening components of changes in the property value.

JBuilder also generates the event-listener registration methods that are called by listening components that want to be notified when the property value of `allTiedUp` changes:

```
public synchronized void removePropertyChangeListener(PropertyChangeListener l) {
    super.removePropertyChangeListener(l);
    propertyChangeListeners.removePropertyChangeListener(l);
}
public synchronized void addPropertyChangeListener(PropertyChangeListener l) {
    super.addPropertyChangeListener(l);
    propertyChangeListeners.addPropertyChangeListener(l);
}
```

The code JBuilder generates for a constrained property is similar except the listeners have the opportunity to reject the change in property value.

## Creating a BeanInfo class

---

You can customize how your bean appears in visual tools such as JBuilder using a `BeanInfo` class. For example, you might want to hide a few properties so they don't appear in JBuilder's Inspector. Such properties can still be accessed programmatically, but the user can't change their values at design time.

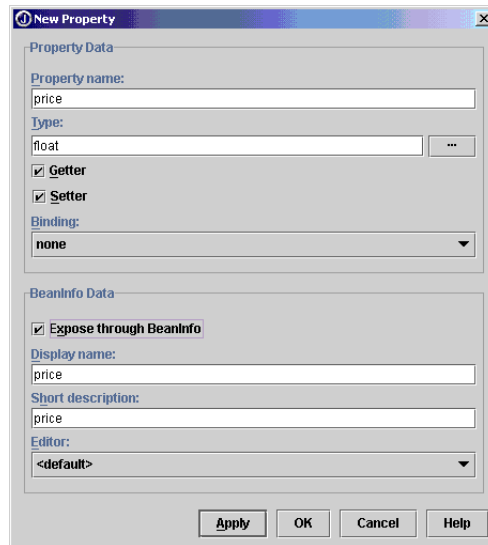
`BeanInfo` classes are optional. You can use `BeansExpress` to generate a `BeanInfo` class for you to customize your bean. If you are going to use a `BeanInfo` class, you should specify more detailed information for each new property you add to your bean.



## Specifying BeanInfo data for a property

---

You can specify BeanInfo data for a property in the New Property dialog box:



To hide a property so it does not appear in visual design tools such as JBuilder's Inspector, make sure the Expose Through BeanInfo option is unchecked.

To provide a localized display name for this property, enter the property name you want to use in the Display Name field.

To provide a short description of this property, enter the description in the Short Description field. JBuilder displays this text as a tool tip in the Inspector for this property.

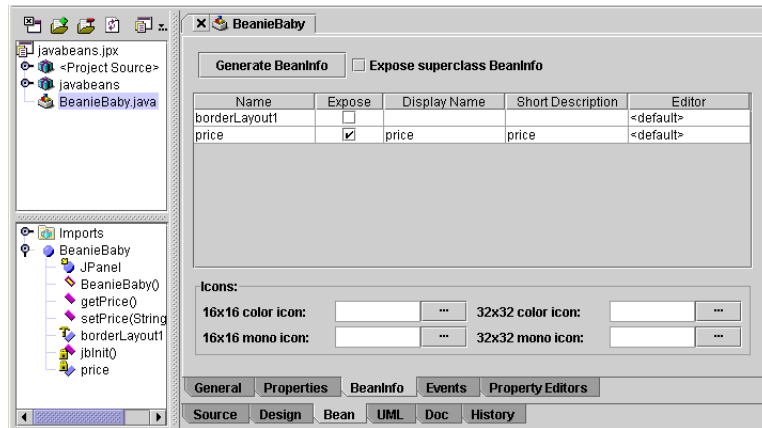
If you have created a property editor that can edit this field, specify the property editor in the Editor field. The Editor field displays all editors in scope that match the given property type.

## Working with the BeanInfo designer

---

The BeanInfo designer provides a way to modify BeanInfo data for a property, lets you specify the icon(s) you want to use to represent your bean in an application builder such as JBuilder, and generates the BeanInfo class for you.

Click the BeanInfo tab of the BeansExpress designer to open the BeanInfo designer.



The BeanInfo designer displays all the properties you have added to your bean. If you change your mind about the BeanInfo data you entered for a property, you can edit one or more of these fields in the grid. Only the name of the property can't be changed.

To provide an image to use as an icon to represent your bean, specify one or more images in the Icons panel.

You can specify different icons to use for both color and mono environments and for different screen resolutions. Fill in the boxes that meet your needs.

If the superclass of your bean has a BeanInfo class and you want its BeanInfo data exposed in the BeanInfo class for your bean, check the Expose Superclass BeanInfo check box. The generated code will include a `getAdditionalBeanInfo()` method that returns the BeanInfo data of the class your bean extends.

To generate a BeanInfo class for your bean, click the Generate Bean Info button.

JBuilder creates a BeanInfo class for you. You'll see it appear in the project pane. To see the generated code, select the new BeanInfo class in the project pane and the source code appears.

## Modifying a BeanInfo class

You can change the BeanInfo class for your bean with BeansExpress.

- 1 Select your bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the BeanInfo tab to display the BeanInfo designer.

- 4 Make your changes in the grid.
- 5 Click the Generate Bean Info button again.

You are warned that you are about to overwrite the existing BeanInfo class. If you choose OK, the class is overwritten with the new BeanInfo data.

## Adding events to your bean

---

A JavaBean can

- Generate (or fire) events, sending an event object to a listening object.
- Listen for events and respond to them when they occur.

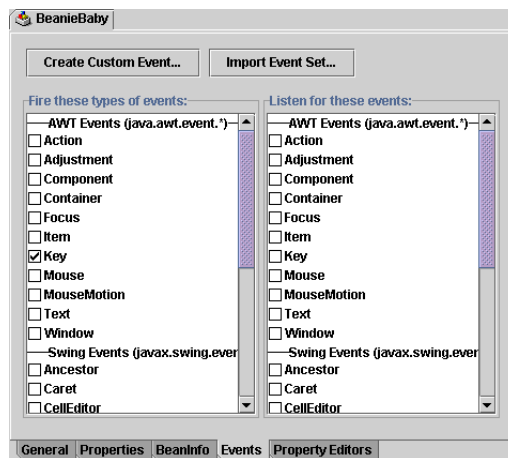
BeansExpress can generate the code that makes your bean capable of doing one or both of these things.

## Firing events

---

To make your bean capable of sending an event object to listening components,

- 1 Select your bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the Events tab to display the Events designer.



- 4 Select the events you want your bean capable of firing in the left window.

The Events designer adds the event-registration methods to your bean. These methods are called by components that want to be notified when

these types of events occur. For example, if you select Key events, these methods are added to your bean:

```
package myjavabean;

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

public class BeanieBaby extends JPanel {
    BorderLayout borderLayout1 = new BorderLayout();
    private float price;
    private transient Vector keyListeners;

    public BeanieBaby() {
        try {
            jbInit();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    private void jbInit() throws Exception {
        this.setLayout(borderLayout1);
    }

    public void setPrice(float price) {
        this.price = price;
    }

    public float getPrice() {
        return price;
    }

    public synchronized void removeKeyListener(KeyListener l) {
        super.removeKeyListener(l);
        if(keyListeners != null && keyListeners.contains(l)) {
            Vector v = (Vector) keyListeners.clone();
            v.removeElement(l);
            keyListeners = v;
        }
    }

    public synchronized void addKeyListener(KeyListener l) {
        super.addKeyListener(l);
        Vector v = keyListeners == null ? new Vector(2) : (Vector)
            keyListeners.clone();
        if(!v.contains(l)) {
            v.addElement(l);
            keyListeners = v;
        }
    }
    ...
}
```

When a component wants to be notified of a key event occurring in your bean, it calls the `addKeyListener()` method of your bean and that component is added as an element in `KeyListeners`. When a key event occurs in your bean, all listening components stored in `KeyListeners` are notified.

The class also generates `fire<event>` methods that send an event to all registered listeners. One such event is generated for each method in the `Listener`'s interface. For example, the `KeyListener` interface has three methods: `keyTyped()`, `keyPressed()`, and `keyReleased()`. So the `Events` designer adds these three `fire<event>` methods to your bean class:

```
protected void fireKeyPressed(KeyEvent e) {
    if(keyListeners != null) {
        Vector listeners = keyListeners;
        int count = listeners.size();
        for (int i = 0; i < count; i++) {
            ((KeyListener) listeners.elementAt(i)).keyPressed(e);
        }
    }
}

protected void fireKeyReleased(KeyEvent e) {
    if(keyListeners != null) {
        Vector listeners = keyListeners;
        int count = listeners.size();
        for (int i = 0; i < count; i++) {
            ((KeyListener) listeners.elementAt(i)).keyReleased(e);
        }
    }
}

protected void fireKeyTyped(KeyEvent e) {
    if(keyListeners != null) {
        Vector listeners = keyListeners;
        int count = listeners.size();
        for (int i = 0; i < count; i++) {
            ((KeyListener) listeners.elementAt(i)).keyTyped(e);
        }
    }
}
```

Once you've made your bean capable of generating events, those events will appear in `JBuilder`'s `Inspector` when the user drops your bean on the `UI` designer.

## Listening for events

---

You can also make your bean a listener for events that occur in other components.

To make your bean a listener, select one of the event types to listen for in the Events designer.

As soon as you check one of the event types, the Events designer implements the associated listener interface in your bean. For example, if you checked `java.awt.event.KeyListener`, the phrase `implements KeyListener` is added to the declaration of your bean and the `KeyPressed()`, `KeyReleased()`, and `KeyTyped()` methods are implemented with empty bodies.

Here is a bean that includes the generated code to implement the `KeyListener` interface:

```
package myBeans;

import java.awt.*;
import javax.swing.JPanel;
import java.beans.*;
import java.awt.event.*;

public class JellyBean extends JPanel implements KeyListener { // implements
                                                                // KeyListener
    BorderLayout borderLayout1 = new BorderLayout();

    public JellyBean() {

        try {
            jbInit();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

    void jbInit() throws Exception {
        this.setLayout (borderLayout1);
    }

    public void keyTyped(KeyEvent e) {           // Adds new method
    }

    public void keyPressed(KeyEvent e) {         // Adds new method
    }

    public void keyReleased(KeyEvent e) {        // Adds new method
    }
}
```

If your bean is registered with another component as a listener (by calling the event-registration method of the component), the source component calls one of the methods in the listening component when that type of event occurs. For example, if a `KeyPressed` event occurs in the source component, the `KeyPressed()` method is called in the listening component. Therefore, if you want your component to respond in some way to such an event, write the code that responds within the body of the `KeyPressed()` method, for example.

## Creating a custom event set

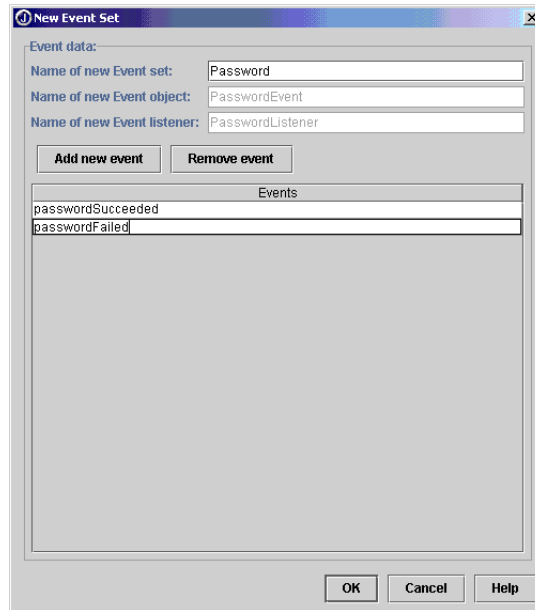
---

Occasionally you might want to create a custom event set to describe other events that can occur in your bean. For example, if your bean implements a password dialog box, you might want the bean to fire an event when a user successfully enters the password. You also might want the bean to fire another event when the user enters the wrong password. Using BeansExpress, you can create the custom event set that handles these situations.

To create a custom event set,

- 1 Select a bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the Events tab.
- 4 Click the Create Custom Event button.  
The New Event Set dialog box appears.
- 5 Specify the name of the event set in the Name Of New Event Set box.  
The names of the event object and the event listener are generated from the name of the event set; they are displayed in the dialog box.
- 6 Select the `dataChanged` item and change it to the name of your first event.

- 7 To add more events, choose the Add New Event button for each event and change the added item to the names of your remaining events:



- 8 Choose OK.

The new event set is added to the list of event types that can be fired in the Events designer.

JBuilder generates the event object class for the event set:

```
package myBeans;

import java.util.*;

public class PasswordEvent extends EventObject {
    public PasswordEvent(Object source) {
        super(source);
    }
}
```

JBuilder also generates the Listener interface for the event set:

```
package myBeans;

import java.util.*;

public interface PasswordListener extends EventListener {
    public void passwordSuccessful(PasswordEvent e);
    public void passwordFailed(PasswordEvent e);
}
```



## Creating a property editor

---

A property editor is an editor for changing property values at design time. You can see several different types of property editors in JBuilder's Inspector. For example, for some properties, you simply type in a value in the Inspector, and by so doing, change the value of the property. This is the simplest type of property editor. For other properties, you use a choice menu (drop-down list) to display all the possible values and you select the value you want from that list. Colors and fonts have property editors that are actually dialog boxes you can use to set their values.

When you create your own JavaBeans, you might create new property classes and want to have editors capable of editing their values. BeansExpress can help you create property editors that display a list of choices.

To begin creating a property editor for a property,

- 1 Click the Property Editors tab in BeansExpress.
- 2 Click the Create Custom Editor button.

The New Property Editor dialog box appears.

- 3 Specify the name of your property editor in the Editor Name box. Give the property editor the same name as the property it will edit with the word Editor appended. For example, if the property name is `favoriteFoods`, name the editor `FavoriteFoodsEditor`.
- 4 Select the type of editor you want from the Editor Type drop-down list.

The appearance of the New Property dialog box changes depending on which type of editor you selected. The next four sections describe how to create a property editor of the four different types.

### Creating a String List editor

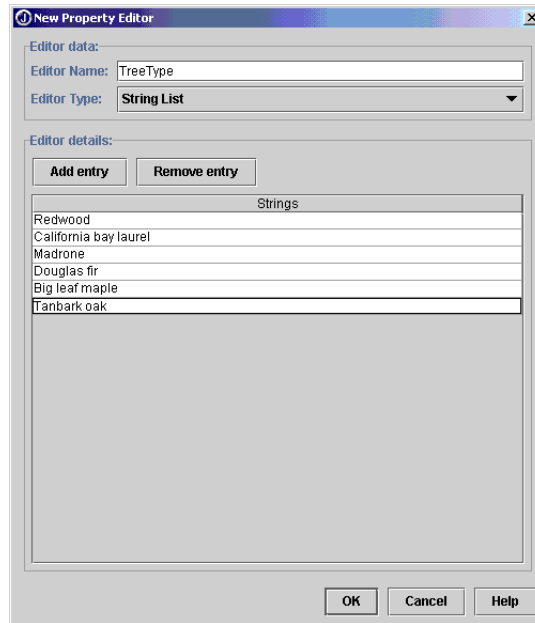
---

A String List is a simple list of Strings. It appears in the Inspector as a drop-down list containing the strings you specify. When the user uses the list to select an entry, the property being edited is set to the selected value.

To add items to a String List editor,

- 1 Choose Add Entry for each item you want to appear in the list.
- 2 In each entry, enter the string you want to appear.

This is how the resulting New Property Editor dialog box might look:



- 3 Choose OK, and a new property editor class is created.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Creating a String Tag List editor

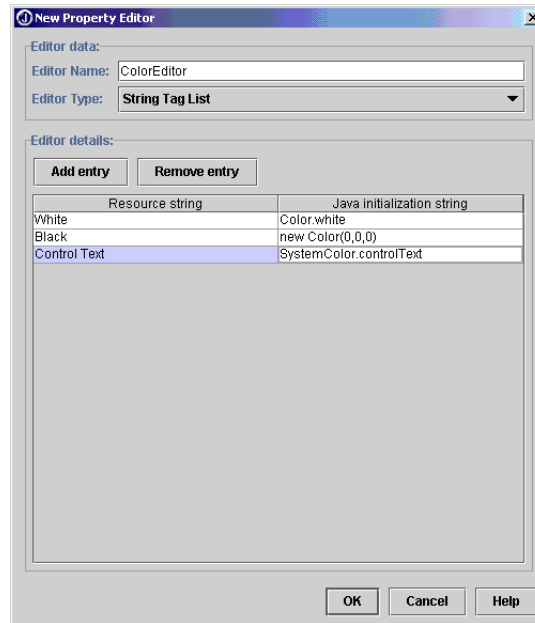
A property editor that is a String Tag List also presents a list of strings to the user in the Inspector. When the user selects one of the items, the specified Java initialization string is used to set the property. A Java initialization string is the string JBuilder uses in the source code it generates for the property.

To add items to a String Tag List editor,

- 1 Choose Add Entry for each item you want to appear in the list.
- 2 In each entry, enter the string you want to appear and its associated Java initialization string.

If you want to include a string in your list of Java initialization strings, put quotation marks (") before and after the string, as if you were entering it in source code.

Here is an example of how the dialog box might look:



- 3 Choose OK, and a new property editor class is created.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Creating an Integer Tag List editor

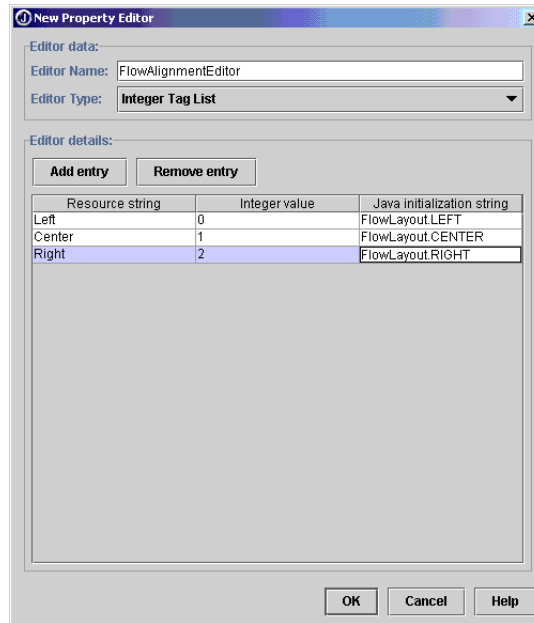
An Integer Tag List property editor can be used to edit integer properties. It presents a list of strings to the user in the Inspector, and when the user selects one of the items, the specified Java initialization string is used to set the property.

To add items to an Integer Tag List editor,

- 1 Choose Add Entry for each item you want to appear in the list.
- 2 In each entry, enter the string you want to appear and its associated integer value and Java initialization string.

If you want to include a string in your list of Java initialization strings, put quotation marks (") before and after the string, as if you were entering it in source code.

Here is an example of how the dialog box might look:



- 3 Choose OK, and a new property editor class is created.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Creating a custom component property editor

You can also use your own custom component to edit the value of a property. Selecting this choice generates a skeleton property editor that uses the custom editor you specify to actually edit the property.

To specify a custom component property editor,

- 1 Enter a name for the editor class that will instantiate your custom component in the Editor Name field of the New Property dialog box.
- 2 Select Custom Editor Component from the drop-down list.
- 3 Specify the name of your custom component as the value of the Custom Component Editor field.
- 4 Check the Support paintValue() option if your custom editor paints itself.

To see the generated code,

- 1 Select your property editor class in the project pane.
- 2 Click the Source tab.

## Adding support for serialization

---

Serializing a bean saves its state as a sequence of bytes that can be sent over a network or saved to a file. BeansExpress can add the support for serialization to your class.

To add support for serialization,

- 1 Select your bean in the project pane.
- 2 Click the Bean tab to display the BeansExpress designers.
- 3 Click the General tab to display the General page.
- 4 Check the Support Serialization option.

BeansExpress modifies the class so that it implements the `Serializable` interface. Two methods are added to the class: `readObject()` and `writeObject()`:

```
void writeObject(ObjectOutputStream oos) throws IOException {
    oos.defaultWriteObject();
}

void readObject(ObjectInputStream ois) throws ClassNotFoundException,
    IOException {
    ois.defaultReadObject();
}
```

You must fill in these two methods to serialize and deserialize your bean.

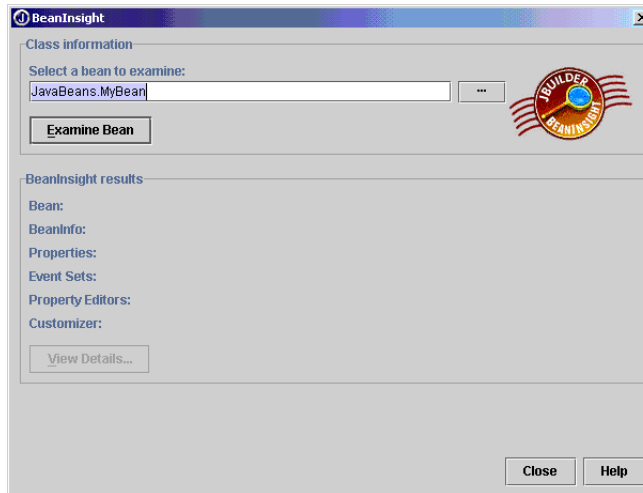
## Checking the validity of a JavaBean

---

When you're finished with your bean, you can use BeanInsight to verify that the component you created is a valid JavaBean component. If your class fails as a JavaBean, BeanInsight reports why it failed. It identifies all the properties, customizers, and property editors it finds for the class. It also reports whether the property information is obtained through a `BeanInfo` class or through introspection.

To verify that your Java class is a JavaBean,

- 1 Choose Tools | BeanInsight to display BeanInsight:



- 2 Type in the name of the component you want examined or use the ... button to specify the component. If the bean is already selected in the project pane when BeanInsight appears, its name is displayed in BeanInsight.
- 3 Click the Examine Bean button to begin the examination process. BeanInsight reports a summary of its findings in the BeanInsight Results section of the wizard.
- 4 To see complete details on BeanInsight's findings, click the View Details button.
- 5 Click the various tabs to see the full report.

## Installing a bean on the component palette

---

Once you have built a valid JavaBean component, you are ready to install it on the component palette using the Palette Properties dialog box. The class files for your new component must be on your classpath.

To display the Palette Properties dialog box, choose Tools | Configure Palette or right-click the component palette and choose Properties.

For information about installing components, click the Help button of the Palette Properties dialog box or see "Adding a component to the component palette" in *Designing Applications with JBuilder*.

# Visualizing code with UML

This is a feature of  
JBuilder Enterprise

*UML*, the *Unified Modeling Language*, is a standard notation for modeling object-oriented systems. UML, at its simplest, is a language that graphically describes a set of elements. At its most complex, it's used to specify, visualize, construct, and document not only software systems but business models and non-software systems. Much like a blueprint for constructing a building, UML provides a graphical representation of a system design that can be essential for communication among team members and to assure architectural soundness of the system.

Using UML for code visualization is a helpful tool for examining code, analyzing application development, and communicating software design. JBuilder uses UML diagrams for visualizing code and browsing classes and packages. UML diagrams can help you quickly grasp the structure of unknown code, recognize areas of over-complexity, and increase your productivity by resolving problems more rapidly.

If you're interested in learning more about UML, visit these web sites:

- Object Management Group at <http://www.omg.org/>
- Cetus UML links at [http://www.cetus-links.org/oo\\_uml.html](http://www.cetus-links.org/oo_uml.html)
- UML Central at [http://www.embarcadero.com/support/uml\\_central.asp](http://www.embarcadero.com/support/uml_central.asp)
- UML Resource Center at <http://www.rational.com/uml/index.jsp>
- UML Zone at <http://www.devx.com/uml/>
- UML Dictionary at <http://softdocwiz.com/UML.htm>

For a tutorial on using JBuilder's UML browser, see [Chapter 20, "Tutorial: Visualizing code with the UML browser."](#)

## Java and UML

---

Because Java and UML are object oriented and platform independent, they work well together. UML, a valuable tool in understanding Java and the complex relationships between classes and packages, assists Java developers in understanding not just a single class but the entire package. In particular, UML can help Java developers who are new to a team get up to speed more quickly on the structure and design of the software system.

### Java and UML terms

---

Because UML is designed to describe a wide range of different scenarios, it uses broad terms to describe different relationships. Listed in the following table are definitions of Java-specific terms and the corresponding UML terms. In some cases, the terms are identical. Throughout this documentation, Java terms are used.

**Table 11.1** Java and UML terms

Java term	Java definition	UML term	UML definition
Inheritance	A mechanism that allows a class or interface to be defined as a specialization of another more general class or interface. For example, a subclass (child) inherits its structure and behavior, such as fields and methods, from its superclass (parent). Classes and interfaces that inherit from a parent use the <code>extends</code> keyword.	Generalization /Specialization	A specialized to generalized relationship in which a specific element incorporates the structure and behavior of a more general element.
Dependency	A using relationship in which a change to an independent object may affect another dependent object.	Dependency	A relationship where the semantic characteristics of one entity rely upon and constrain the semantic characteristics of another entity.
Association	A specialized dependency where a reference to another class is stored.	Relationship (Association)	A structural relationship that describes links between or among objects.
Interface	A group of constants and method declarations that define the form of a class but do not provide any implementation of the methods. An interface specifies what a class must do but not how it gets done. Classes and interfaces that implement the interface use the <code>implements</code> keyword.	Realization / Interface	A collection of operations used to specify a service of a class or component. States the behavior of an abstraction without the implementation of that behavior.



**Table 11.1** Java and UML terms (continued)

Java term	Java definition	UML term	UML definition
Method	The implementation of an operation which is defined by an interface or class.	Operation	An implementation of a service that can be requested by an object and can affect that object's behavior. Operations are usually listed below the attributes in a UML diagram.
Field	An instance variable or data member of an object.		
Property	Information about the current state of a component. Properties can be thought of as named attributes of a component that a user or program can read (get) or write (set). In a UML diagram, a property exists when a field name matches a method name which is preceded by "is", "set", or "get". For example, a field named <code>parameterRow</code> is a property if it has a method named <code>setParameterRow()</code> .	Attribute	A named property of a classifier, such as a class or an interface, that describes values that instances of a property can hold.

## JBuilder and UML

JBuilder focuses on code visualization and UML diagramming specific to the Java language, rather than replacing the many available UML design tools. UML functionality in JBuilder allows you to visually browse packages and classes to help you better design, understand, and troubleshoot your application development process.

Two UML diagrams are available in JBuilder:

- Limited package dependency diagrams
- Combined class diagrams

JBuilder's UML browser also provides additional features, such as refactoring code, customizing the UML display, as well as viewing Javadoc and source code.

**Important** In obfuscated code, JBuilder excludes private fields and members in a class file.

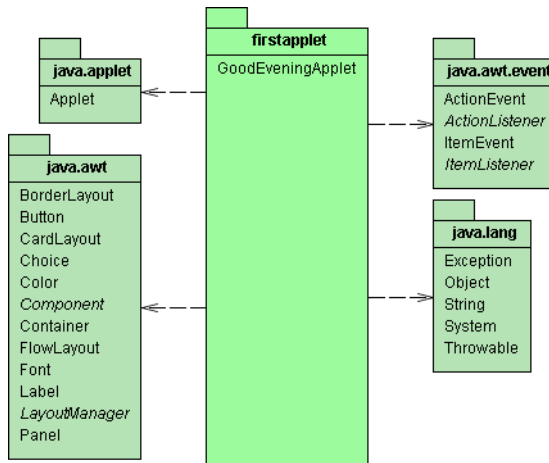
### See also

- ["Customizing UML diagrams" on page 11-17](#)
- [Chapter 12, "Refactoring code symbols"](#)

## Limited package dependency diagram

The *package diagram* is centered around a central package and shows only the dependencies of that package. It does not show the dependencies between the dependent packages. Dependencies and reverse dependencies appear on the left side, the right side, or both sides of the central package. Packages with reverse relationships are sorted to either side, mixed relations in the middle, and the remainder below. Packages with dependencies display the specific dependent classes within the package, which can be used for navigation by double-clicking them in the diagram. The current, central package is displayed in a bright green background by default. All other packages have a darker green background by default.

**Figure 11.1** Package diagram



Although only the current package and imported packages are displayed in the UML diagram, you can include references from the project library classes. To include library references, set the Include References From Project Library Class Files option on the General page of the Project Properties dialog box (Project | Project Properties).

### See also

- [“Viewing package diagrams” on page 11-12](#)
- [“Including references from project libraries” on page 11-18](#)

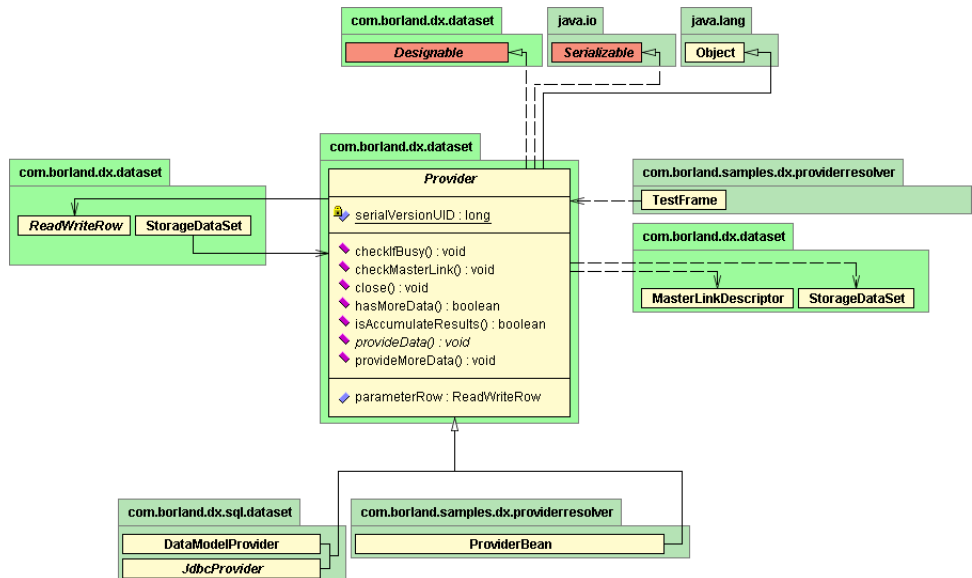
## Combined class diagram

The *combined class diagram* for a Java source file or class file open in the editor displays the class in the center of the diagram with associations on

the left and dependencies on the right. Extended classes (superclasses) and extended interfaces (parent interfaces) appear on the top, while extending and implementing classes appear on the bottom. Classes are grouped according to package.

Grouped associations and dependencies are sorted as follows: reverse associations are on the top left side of the central class and reverse dependencies are on the top right; associations and dependencies with mixed relations are on the middle left and middle right of the class; all remaining associations and dependencies are on the lower left and lower right of the class.

**Figure 11.2** Class diagram



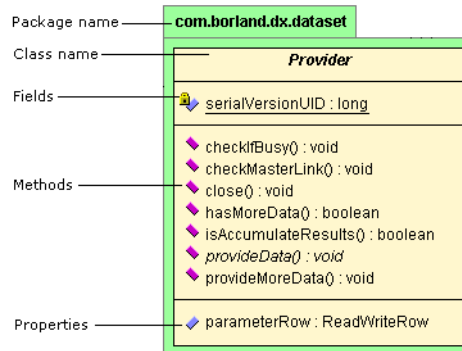
The UML class diagram displays the class in the center of the diagram in a rectangle with a default yellow background. Surrounding the class is the package with the package name in a tab at the top. The class itself is divided into several sections, which are separated by horizontal lines, in the following order:

- Class name at the top
- Fields and properties\*
- Methods, getters\*, and setters\*
- Properties\* at the bottom

\*By default, properties are displayed in the bottom section of the class diagram. The Display Properties Separately option is set on the UML page of the IDE Options dialog box (Tools | IDE Options). If this option is

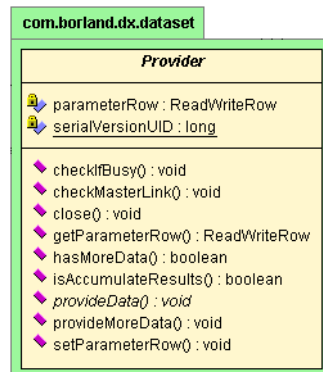
turned off, properties are displayed in the appropriate sections with fields and methods. See [“Setting IDE Options” on page 11-19](#).

**Figure 11.3** Class diagram with properties displayed separately



**Note** Icons indicate whether a field, method, or property is private, public, or protected. See [“Visibility icons” on page 11-9](#).

**Figure 11.4** Class diagram without properties displayed separately



Although only the current package and imported packages are displayed in the UML diagram, you can include references from the project library classes. To include library references, set the Include References From Project Library Class Files option on the General page of the Project Properties dialog box (Project | Project Properties).


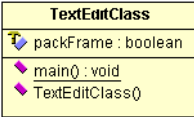
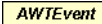




### See also

- [“Viewing class diagrams” on page 11-12](#)
- [“Including references from project libraries” on page 11-18](#)




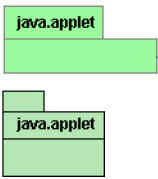
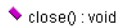
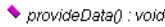
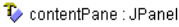
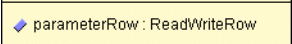

## JBuilder UML diagrams defined

The following table lists definitions for folders in the structure pane, terms in the diagrams, and the corresponding UML representation. For example, dependencies appear in a Dependencies folder in the structure pane and are represented in the UML diagram by a dashed line.

**Table 11.2** UML diagram definitions

Diagram	Definition	Diagram Description	Diagram Example
Extended Classes	Classes whose attributes (fields and properties) and methods are inherited by another class. Also called superclass, parent class, or base class.	A solid line with a large triangle that points from the subclass (child class) to the superclass (parent class). Displayed at the top of the UML diagram.	
Classes	Structures that define objects. A class definition defines fields and methods.	Displayed in a rectangular box with a default yellow background with the name at the top and fields, methods, and properties listed below it.	
Abstract classes	Classes that are superclasses of another class but that can't be instantiated.	Displayed in italic font.	
Extending Classes	Classes that extend (inherit from) the superclass. Also called subclass or child class.	A solid line with a large triangle that points from the subclass to the superclass. Displayed at the bottom of the UML diagram.	
Implementing Classes	Classes that implement the central interface.	A dashed line with a large triangle which points from the implementing class to the inherited interface. Displayed at the bottom of the UML diagram.	
Extended Interfaces	Parent interfaces that are inherited by a subinterface.	A solid line with a large triangle that points from the subinterface to the inherited interface. Displayed at the top of the UML diagram.	
Interfaces	Groups of constants and method declarations that define the form of a class but do not provide any implementation of the methods. Interfaces allows you to specify what a class must do but not how it gets done.	A rectangle with a default orange background and the interface name in italic font.	

**Table 11.2** UML diagram definitions (continued)

Diagram	Definition	Diagram Description	Diagram Example
Implemented Interfaces	Interfaces that are implemented by the central class.	A dashed line with a large triangle which points from the implementing class to the implemented interface. Displayed at the top of the UML diagram.	
Dependencies/Reverse Dependencies	Using relationships in which a change to the used object may affect the using object.	A dashed line with an arrowhead.	
Associations/Reverse Associations	Specialized dependencies where a reference to another class is stored.	A solid line with an arrowhead.	
Packages	Collections of related classes.	A rectangle with a tab at the top and the package name in the tab or below it. The current package has a bright green background by default. All other packages have a darker green background by default.	
Methods	Operations defined in a class or interface.	Listed below members and fields, including the return type.	
Abstract methods	Methods that don't have any implementation.	Displayed in italic font.	
Members/fields	Instance variables or data members of an object.	Listed below the class name, including the return type.	
Properties	Properties exist when a method name matching a field name is preceded by "is", "get", or "set". For example, a field name <code>parameterRow</code> with a <code>getParameterRow()</code> method is a property.	Properties are displayed separately in the bottom section of the class diagram if the Display Properties Separately option is set on the UML page of the IDE Options dialog box (Tools   IDE Options).	
Static	Having class scope.	Static members, fields, variables, and methods are underlined in the UML diagram.	

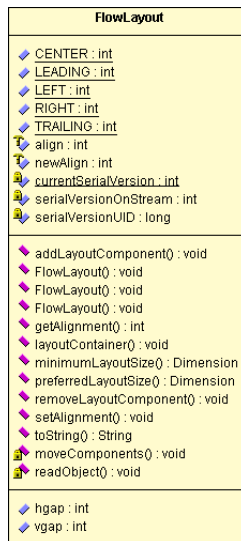
## Visibility icons

UML uses icons to represent the visibility of a class, such as `public`, `private`, `protected`, and `package`. You can use JBuilder's visibility icons or the standard UML icons in your UML diagrams.

JBuilder's visibility icons are the same icons used in the structure pane in the source code. To use JBuilder's icons in your UML diagrams, choose the Use Visibility Icons option on the UML page of the IDE Options dialog box (Tools | IDE Options). The Use Visibility Icons option is on by default.

**Note** For structure pane icon definitions, see “JBuilder structure pane and UML icons” in *Introducing JBuilder*.

**Figure 11.5** JBuilder's visibility icons



UML uses more generic icons to represent the visibility of a class as defined in the following table.

**Table 11.3** UML visibility icons

UML icon	Description
+	public
-	private
#	protected

To use the standard UML visibility icons in the UML diagram, uncheck the Use Visibility Icons option on the UML page of the IDE Options dialog box (Tools | IDE Options).

FlowLayout
<ul style="list-style-type: none"> <li>+ CENTER : int</li> <li>+ LEADING : int</li> <li>+ LEFT : int</li> <li>+ RIGHT : int</li> <li>+ TRAILING : int</li> <li>+ align : int</li> <li>+ newAlign : int</li> <li>- currentSerialVersion : int</li> <li>- serialVersionOnStream : int</li> <li>- serialVersionUID : long</li> </ul>
<ul style="list-style-type: none"> <li>+ addLayoutComponent() : void</li> <li>+ FlowLayout() : void</li> <li>+ FlowLayout() : void</li> <li>+ FlowLayout() : void</li> <li>+ getAlignment() : int</li> <li>+ layoutContainer() : void</li> <li>+ minimumLayoutSize() : Dimension</li> <li>+ preferredLayoutSize() : Dimension</li> <li>+ removeLayoutComponent() : void</li> <li>+ setAlignment() : void</li> <li>+ toString() : String</li> <li>- moveComponents() : void</li> <li>- readObject() : void</li> </ul>
<ul style="list-style-type: none"> <li>+ hgap : int</li> <li>+ vgap : int</li> </ul>

## Viewing UML diagrams

---

JBuilder provides a UML browser for visualizing your code using UML diagrams. The UML browser, available on the UML tab of the content pane, displays package and class diagrams using standard UML. When you choose the UML tab, JBuilder loads the class files to determine their relationships, which the UML browser then uses to obtain the package and class information for the UML diagrams.

For an up-to-date and accurate UML diagram, it's always best to compile before you choose the UML tab. The UML browser does display Java source files dynamically even if they haven't been compiled, but only if they are on the source path. A message displays in the UML browser indicating that the UML diagram may not be accurate. However, if a source file isn't on the source path, the `.class` file must be generated first. A message prompts you to compile the project to generate the class files for the UML diagram. If the class files are out of date, for example the source file has been changed but hasn't been recompiled, a message displays in the UML browser indicating that the UML diagram may not be accurate.

The UML browser also supports diagramming of reverse dependencies from classes to JSPs (Java ServerPages). For example, a bean generated by the JSP wizard links to the JSP that uses it. It doesn't have to be a JSP bean; it could be any class that the JSP uses.



JBuilder doesn't include any references from project libraries in the UML diagram unless you choose the Include References From Project Library Class Files on the General page of the Project Properties dialog box (Project | Project Properties). See [“Including references from project libraries” on page 11-18](#).

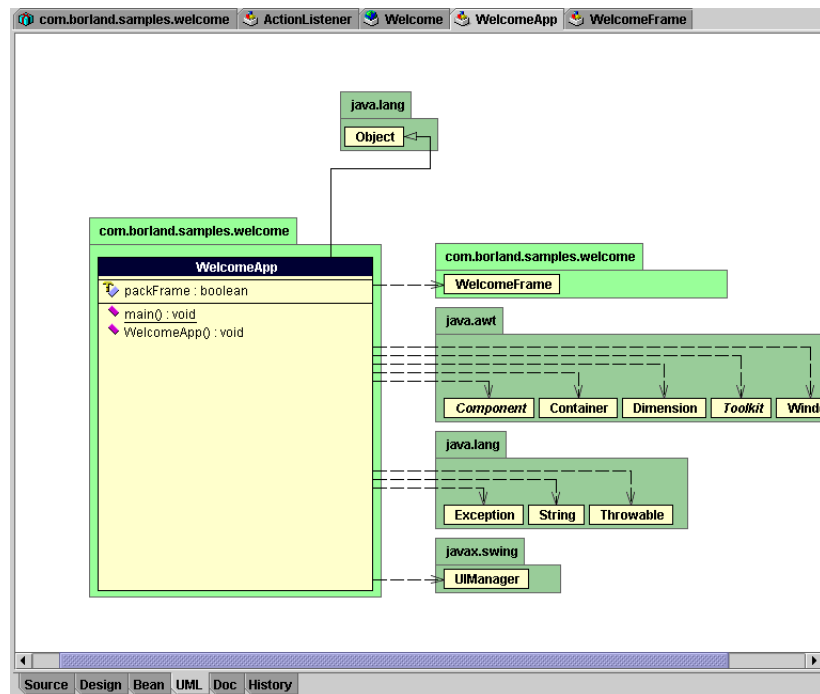
## JBuilder's UML browser

JBuilder's UML browser provides various features for customizing the diagram display, navigating diagrams and source code, viewing inner classes, source code, and Javadoc, creating and printing images, and refactoring.

You can view a UML diagram in JBuilder by opening a package or class and selecting the UML tab in the content pane.

**Note** If your project is large, it may take some time to view the UML diagram for the first time. JBuilder needs to load the classes to determine their relationships before building the diagrams.

**Figure 11.6** UML browser



## Viewing package diagrams

---

To view a limited package dependency diagram,

- 1 Choose Project | Make Project or Project | Rebuild Project to compile the project.
- 2 Double-click the package node in the project pane or right-click it and select Open.
- 3 Choose the UML tab in the content pane to view the package diagram.

**Note** If the package node is not available, choose Project | Project Properties | General and check the Enable Source Package Discovery And Compilation option.

## Viewing class diagrams

---

To view a combined class diagram,

- 1 Choose Project | Make to compile your project or Java file.
- 2 Double-click a Java file in the project pane to open it or right-click it and choose Open.
- 3 Choose the UML tab at the bottom of the content pane to view the UML class diagram.

## Viewing inner classes

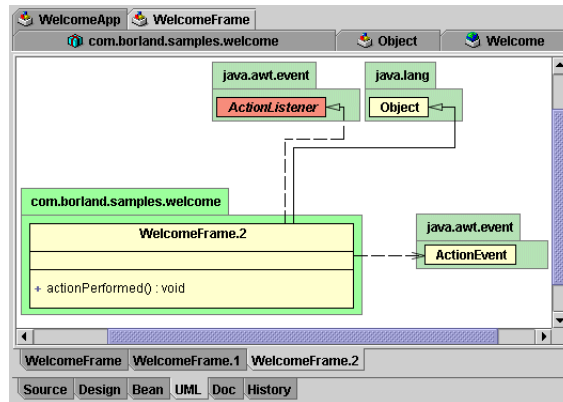
---

A single class may contain more than one class, including inner classes and anonymous inner classes. In this case, the UML browser presents a tabbed user interface with one class per tab.

Individual anonymous inner classes are only diagrammed if the editor cursor is positioned in that class or the class is navigated to from another diagram. To position the cursor in the editor, choose the Source tab on an open source file, position the cursor, and choose the UML tab.

Such selected anonymous inner classes are remembered until the file is closed, so they can accumulate as tabs in the UML browser. Dependencies of anonymous inner classes are folded into the classes which contain them.

The UML browser uses the cursor position in the editor to determine the class, method, and/or field that is selected in the UML browser. However, if that cursor position is unchanged on subsequent visits to the viewer, the last selection is retained. Note that for fields and methods, the cursor has to be between the first and last character of the definition and not at the start of the line.

**Figure 11.7** Viewing inner classes

## Viewing source code

In a class diagram, you can navigate to the source code and back to the UML diagram. Double-click the central class, a method, a field, or a property to view the source file for the class. The cursor is positioned appropriately in the editor. Conversely, positioning the cursor in a class, method, field, or property in the editor also highlights it in the UML diagram. When in the editor, choose the UML tab to return to the UML diagram.

The UML browser has a menu selection on the context menu: Go To Source. Choose Go To Source to see the source code in the editor. This can be useful for viewing source code for other classes and interfaces in package and class diagrams.

The UML browser also provides tool tips for quickly viewing the argument list for methods. Move the mouse over a method to see its tool tip. Move the mouse over a class name to see its fully qualified class name, which includes the package name.

## Viewing Javadoc

There are several ways to access Javadoc for packages, interfaces, classes, methods, and fields within a UML diagram.

- Select an element in the UML diagram, right-click, and choose View Javadoc.
- Select an element in the UML diagram and press *F1*.
- Select an element in the structure pane and press *F1*.

JBuilder's Help Viewer displays the Javadoc, which is generated either from Javadoc comments in the source file or from available information, such as method signatures.

**Note** If you've run Javadoc with the **javadoc** tool or the Javadoc wizard, more information is included.

### See also

- [“Viewing Javadoc” on page 14-20](#)

## Using the context menu

---

The UML browser has a context menu for quickly accessing common commands. Right-click an element in the UML browser to activate the menu. See the following documentation for information on these commands and what they do:

- Find References: [“Learning about a symbol before refactoring” on page 12-7](#)
- Rename: [“Rename refactoring” on page 12-2](#)
- Move: [“Move refactoring” on page 12-3](#)
- Change Parameters: [“Changing method parameters” on page 12-22](#)
- Save Diagram: [“Creating images of UML diagrams” on page 11-20](#)
- Enable Class Filtering: [“Filtering packages and classes” on page 11-18](#)
- Go To Diagram: [“Navigating diagrams” on page 11-15](#)
- Go To Source: [“Viewing source code” on page 11-13](#)
- View Javadoc: [“Viewing Javadoc” on page 11-13](#)

## Scrolling the view

---

There are several ways to scroll the UML diagram in the UML browser:

- Click and drag with the mouse
- Up and Down keys
- Arrow keys
- Scroll bars

The behavior differs according to the type of view displayed. View | Hide All is the full view and only displays the content pane. View | Show All displays the following panes by default: project pane, content pane, and structure pane.

## Full view

In the full view (View | Hide All), you can use the mouse to move the view up and down. Select the background of the diagram, then click, and drag the diagram. The *Page Up* and *Page Down* keys, as well as the up and down arrow keys, also move the view up and down. You can also manually scroll the view using the scroll bars.

## Partial view

In the partial view (View | Show All), you can drag the diagram in all directions. Select the background of the diagram, then click and drag the diagram in any direction. The *Page Up* and *Page Down* keys move the view up and down. All four arrow keys can also be used to move the view in any direction. You can also manually scroll the view using the scroll bars.

## Refreshing the view

---

To refresh the UML diagrams after making changes to your project, use one of these methods:

- Rebuild your project (Project | Rebuild Project).
- Press the Refresh button on the project pane toolbar.



# Navigating diagrams

---

Double-click a package or class name in the UML diagram to view its UML diagram. When an element is selected in the UML diagram, the background highlighting color changes. After selection, you can use the *Arrow* keys to move up and down the diagram. If nothing is selected, the *Page Up* and *Page Down* keys scroll the diagram up and down. To browse previously viewed UML diagrams, use the browser history Forward and Back buttons available on the main toolbar for easy back and forth navigation between UML diagrams.

You can also navigate by choosing packages and classes in the structure pane. Click a package or class to select it in the diagram. Double-click a class to see its diagram. Right-click a package and choose Open to view the package diagram.

There's also a selection on the UML browser's context menu for viewing UML diagrams: Go To Diagram. Right-click a package, class, or interface name in the UML diagram and choose Go To Diagram to view its diagram.

## See also

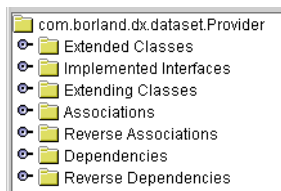
- [“UML and the structure pane” on page 11-16](#)

## UML and the structure pane

The structure pane, located to the lower left of the UML diagram, provides a tree view of relationships contained in expandable folders by category, such as Extended Classes and Dependencies. If any of the categories are not included in the diagram, the folder doesn't appear. These folders offer navigation to other diagrams and may also provide information which is not in the diagram, since they reflect the relationships without regard to filtering settings or other restrictions. For example, even if you have filtered out specific classes and packages, they still appear in the structure pane. See [“Customizing UML diagrams” on page 11-17](#) for more information on filtering UML diagrams. To expand and collapse folder icons in the structure pane, double-click them or toggle the expand icon.

Packages and classes that aren't shown in the diagram display in a lighter color in the structure pane. They aren't displayed if they're filtered out or if they're redundant and removed for clarity.

**Figure 11.8** Structure pane for UML diagrams



The structure pane can also be used for selection and navigation to a class or package. Select a class, interface, or package in the structure pane to select it in the diagram. Double-click a class or package in the structure pane to navigate to its UML diagram. Right-click a package and choose Open to view the UML package diagram.

You can quickly search for a package or class in the structure pane by moving the focus to the tree and starting to type the name you want. For more information, see “Searching trees” in “The JBuilder environment” chapter of *Introducing JBuilder*.

### Package diagrams

For package diagrams, the folders can include any or all of the following:

- Dependencies
- Reverse Dependencies

For definitions of these terms, see [“JBuilder UML diagrams defined” on page 11-7](#).

Opening a dependent package shows all the classes in that package with the given relationship to the central package. This allows you to find out which classes in a dependent package are causing the dependency.

## Class diagrams

For class diagrams, the folders can include any or all of the following:

- Extended Classes
- Extended Interfaces
- Implemented Interfaces
- Extending Classes
- Implementing Classes
- Associations
- Reverse Associations
- Dependencies
- Reverse Dependencies

For definitions of these terms, see [“JBuilder UML diagrams defined” on page 11-7](#).

## Customizing UML diagrams

---

Although you can't manipulate the UML diagrams, such as moving or resizing elements, you can customize the UML display in the Project Properties and the IDE Options dialog boxes. For example, you can filter what is displayed in a given diagram on a project basis, as well as include references from project libraries. You can also globally customize the display of your UML diagram by setting the sort order, font, colors, and various other options.

## Setting project properties

---

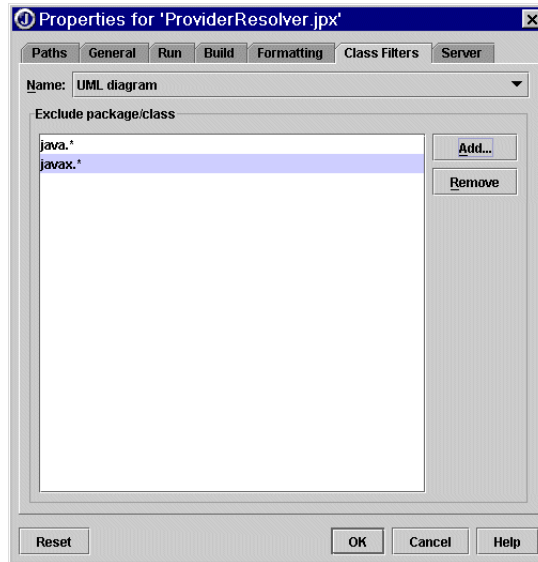
There are several project properties you can set for your UML diagrams in the Project Properties dialog box:

- Filtering: available on the Class Filters page
- Library references: available on the General page
- References from generated source: available on the General page

To open the Project Properties dialog box, choose Project | Project Properties or right-click the project file in the project pane and choose Properties.

## Filtering packages and classes

On the Class Filters page of the Project Properties dialog box, you can exclude packages and classes from your project's UML diagrams. Choose Project | Project Properties and click the Class Filters tab. Choose UML Diagram from the Name drop-down list. Then choose the Add button to add any classes or packages to the exclusion list. Any classes or packages in the list are then excluded from the UML diagram.



Return to your UML diagram and choose the Refresh button on the project pane toolbar. Notice that the classes and packages in the Exclude Package/Class list are excluded from the diagram but are still accessible in the structure pane. To temporarily disable any package and class filtering applied to your diagram, right-click the UML diagram and uncheck Enable Class Filtering on the UML browser context menu.

### Important

If you have filtering set in the Project Properties dialog box, all of the diagrams in the project are filtered. Disabling filtering from the context menu in one diagram does not disable it for all diagrams. If you navigate to another diagram in the project, filtering is still enabled. Once you close the file or package, the setting reverts back to the project-level setting.

## Including references from project libraries

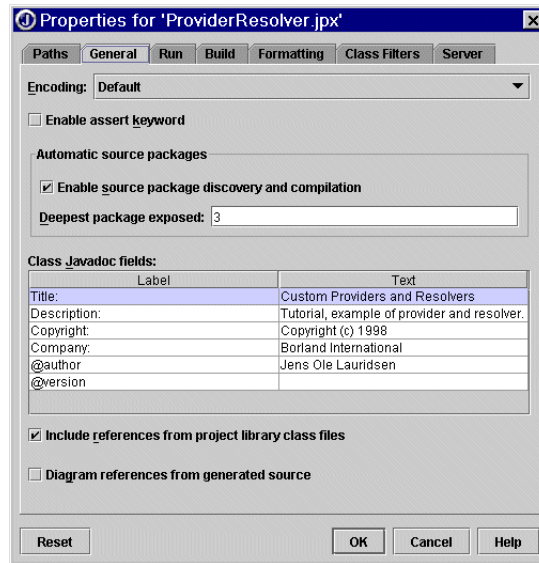
Typically, libraries provide services to the applications that are built upon them but don't know anything about their users. To show these relationships, you need to include references from the libraries.

On the General page of the Project Properties, you can check the Include References From Project Library Class Files option to include references



from libraries in your UML diagrams. By default, this option is off and a library's reverse dependencies to a project are excluded.

**Note** If your project is very large, choosing this option could noticeably increase the time it takes JBuilder to load the classes and create the UML diagram.



### See also

- [“Step 4: Adding references from libraries” on page 20-9 in Chapter 20, “Tutorial: Visualizing code with the UML browser.”](#)

### Including references from generated source

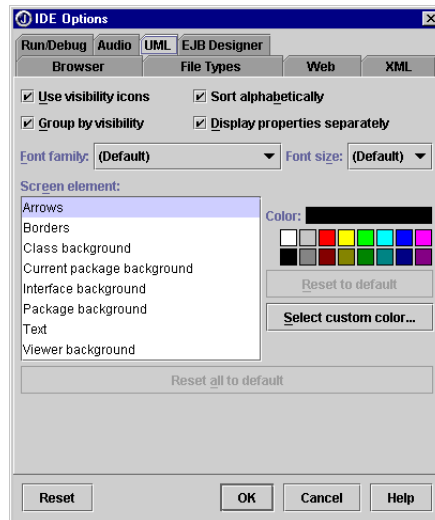
You can also include references from generated source, such as IIOP files and EJB stubs, in your UML diagrams. To do this, choose the Diagram References From Generated Source option on the General page of Project Properties.

## Setting IDE Options

The UML page of the IDE Options dialog box (Tools | IDE Options) provides options for global customization of the UML diagram in JBuilder's UML browser. To access the UML page, choose Tools | IDE Options, and click the UML tab.

Here you can change the UML diagram's visibility icons, grouping order, sorting order, properties display, font family and size, and colors for the

various screen elements. Choose the Help button on the UML page for more information.



### See also

- “Customizing the IDE” in *Introducing JBuilder*

## Creating images of UML diagrams

---

The UML browser supports saving UML diagrams as images. However, the image size and color depth may be a limitation. In that case, an error message is output. JBuilder supports the Portable Network Graphics (PNG) format for images.

To save the UML diagram as an image, right-click in the UML browser and choose Save Diagram. Enter a file name in the Save Diagram dialog box. The PNG extension is automatically added to the file name.

## Printing UML diagrams

---

Use the Print button on the main toolbar or the Print command (File | Print) to print your UML diagram. You can also use the Page Layout command (File | Page Layout) to set up page headers, set margins, and change the page orientation. The diagram is scaled down slightly from the size on the screen. Diagrams that are too large to fit on a page are printed as multiple pages.

**Important** You need to move the focus to the UML diagram for the print option to be available.

## Refactoring and Find References

---

The UML browser provides access to JBuilder refactoring features. There are several ways to access refactoring in the UML browser:

- Right-click a package, class, field, method, or property name in the UML diagram and choose Rename from the context menu.
- Select a package, class, field, or method name in the UML diagram and press *Enter*. Enter a new name in the Rename dialog box.
- Right-click a class name in the UML diagram and choose Move from the context menu.
- Right-click a method name in the UML diagram and choose Change Parameters from the context menu.

Before refactoring, you might also want to find all source files using a selected symbol. To locate all references to a symbol, select the symbol in a UML diagram or in the editor. Right-click the symbol and choose Find References.

### See also

- [Chapter 12, “Refactoring code symbols”](#)
- [“Finding references to a symbol” on page 12-8](#)



# Refactoring code symbols

This is a feature of  
JBuilder SE and  
Enterprise

Code evolves as technology and market demands evolve. Over time, existing code may need to be changed to allow more room to grow, to improve performance, to accommodate changing needs, or simply to clean up the code base. *Refactoring* is the term used to describe redesigning existing code without changing its behavior from the user's point of view. It also means the individual tasks involved in that redesign.

Refactorings may be small or extensive, but even small changes can introduce bugs. Refactoring must be done correctly and completely to be effective. One change can have permutations throughout the entire codebase. Good refactoring handles the entire set of permutations responsibly and durably, so that no behavior is changed beyond improvements in performance or maintainability and no bugs are introduced.

## Types of refactorings

---

JBuilder provides many types of refactorings:

- Optimize Imports
- Rename refactoring
- Move refactoring
- Change Parameters
- Extract Method
- Introduce Variable
- Surround With Try/Catch

Refactorings are available from the editor context menu, the Editor menu, the structure pane context menu, and a UML diagram context menu. Note that you cannot refactor across projects.

**Note** If you refactor an EJB file, you will see the following warning:

WARNING: You are refactoring an EJB file. This may require that you change some source code and the deployment descriptor by hand. We recommend using the EJB Designer for most refactoring scenarios.

You will need to update all relevant source files to support the refactoring. (EJB development is a feature of JBuilder Enterprise.)

## Optimize Imports

---

An Optimize Imports refactoring rewrites and reorganizes your `import` statements according to the custom settings in the project properties. It also removes any `import` statements that are no longer used in the code.

To learn how to optimize imports in JBuilder, see [“Optimizing imports” on page 12-14](#).

## Rename refactoring

---

Rename refactoring applies a new name to a package, class, method, field, local variable, or property, ensuring that all references to that name are correctly handled. Rename refactoring a constructor renames the class.

Rename refactoring is far more than a search and replace task; references must all be accounted for and properly handled while patterns must be recognized so that overloaded names are handled correctly. For example, when a rename refactoring is performed on an overloaded class name, the class’s new name must be reflected in the class declaration and in every instance of that class and every other reference to that class. However, the new name must only be reflected in the target class, not in the other classes that share its original name or their declarations, instances, references, etc.

For packages, rename refactoring renames the specified package. Package and import statements in class files are updated. The package, sub-packages, and class source files are moved to the new source directory and the old one is deleted.

In JBuilder, you can rename refactor the following code symbols.

**Table 12.1** Refactoring and code symbols

Code symbol	Description
Package	Rename refactoring a package renames the package and the entire sub-tree of packages. The package name cannot already exist in the project.
Class, inner class, or interface	Rename refactoring an outer public class renames the source file. If the source file name already exists in the current package, the refactoring is prevented. If the class is not the outer public class and there is another class of the desired new name, the class isn't renamed.
Method	Rename refactoring a method renames the method and all references to that method. The method can be renamed in all classes that this class inherits from or in all classes in the hierarchy for the class. A forwarding method can be created.
Field	Rename refactoring a field renames the field to a new name. The new name cannot already exist in the class that declared the original name. If there are scope conflicts between the new name and the old name, the <code>this</code> keyword is added to the beginning of the new field name. A warning is displayed if the new name overrides or is overridden by an existing field in a superclass or subclass.
Local variable	Rename refactoring a local variable renames the variable to the new name. The new name cannot already exist in the class that declared the original name. The local variable name is prepended to a field name if there is a conflict with a new variable name.
Property	Rename refactoring a property renames the property, as well as its getter and setter. The new name cannot already exist in the class that declared the original name.

To learn how to rename refactor in JBuilder, see:

- [“Rename refactoring a package” on page 12-17](#)
- [“Rename refactoring a class” on page 12-17](#)
- [“Rename refactoring a method” on page 12-19](#)
- [“Rename refactoring a field” on page 12-21](#)
- [“Rename refactoring a local variable” on page 12-20](#)
- [“Rename refactoring a property” on page 12-22](#)

## Move refactoring

In JBuilder, move refactoring is available for classes. Move refactoring moves a specified class to a new package. Move refactoring is only allowed on a top-level public class. The package the class is being moved to cannot already contain a source file of the new name. The refactoring

must update the declaration of the class, as well as all the usages of that class.

To learn how to move refactor in JBuilder, see [“Move refactoring a class” on page 12-18.](#)

## Change Parameters

---

Change Parameters allows you to add, rename, delete and re-order a method's parameters. You can edit a newly added parameter before you close the dialog; however, you cannot edit an existing parameter.

To learn how to change method parameters, see [“Changing method parameters” on page 12-22.](#)

## Extract Method

---

Extract Method turns a selected code fragment into a method. JBuilder moves the extracted code outside of the current method, determines the needed parameter(s), generates local variables if necessary, and determines the return type. It inserts a call to the new method in the code where the code fragment resided.

To learn how to extract a method, see [“Extracting a method” on page 12-24.](#)

## Introduce Variable

---

The Introduce Variable refactoring allows you to replace the result of a complex expression, or part of the expression, with a temporary variable name that explains the purpose of the expression or sub-expression.

To learn how to introduce a variable in JBuilder, see [“Introducing a variable” on page 12-25.](#)

## Surround With Try/Catch

---

This refactoring adds a `try/catch` statement around the selected block of code. It detects all checked exceptions in a block and adds specific blocks for each checked exception.

To learn how to surround a code block with a `try/catch` statement, see [“Surrounding a block with try/catch” on page 12-26.](#)



## JBuilder's refactoring tools

---

You can access JBuilder's refactoring tools from the editor context menu and a UML diagram context menu. Refactoring commands are also available on the Edit menu and the Search menu. For more information on the editor, see "Working in the editor" in *Introducing JBuilder*. For more information on UML, see [Chapter 11, "Visualizing code with UML."](#) (UML is a feature of JBuilder Enterprise.)

Before a refactoring, you can view, by category, all locations in the current project where the selected symbol is referenced. You can also navigate to the symbol's definition. If JBuilder can't complete the refactoring, the IDE provides warning and error messages to help explain why. Warnings don't stop the refactoring. However, if an error is encountered, the refactoring is prevented. For example, a refactoring might be prevented if a file is read-only (not yet checked out) or if the symbol name already exists.

**Note** Single file refactorings (for example, Extract Method and Introduce Variable) do not display output unless there are errors or warnings.

JBuilder's refactoring tools provide extensive information, including:

- Limitations reporting

JBuilder checks for conditions where your refactoring might encounter problems. For example, JBuilder determines if needed dependency information is missing or out-of-date, if a file is read-only, or if a class file does not exist.

- References discovery

JBuilder finds all source files containing dependencies. The exact source position is located.

- Validation

JBuilder determines if the new name is legal. For example, the name might already be in use or contain illegal syntax.

- Source tree updating

JBuilder physically moves a directory or a file within the source tree for a class move refactoring or a package rename refactoring. JBuilder also updates import statements as needed for any dependencies.

- Reference renaming

JBuilder renames references with the new name.

## Setting up for references discovery and refactoring

To find all references to a symbol, you need to compile with references from project libraries enabled. To verify this, go to the General tab of the Project Properties dialog box. Make sure the Include References From Project Library Class Files option is on. This option loads all library relationships, allowing JBuilder to discover all references.

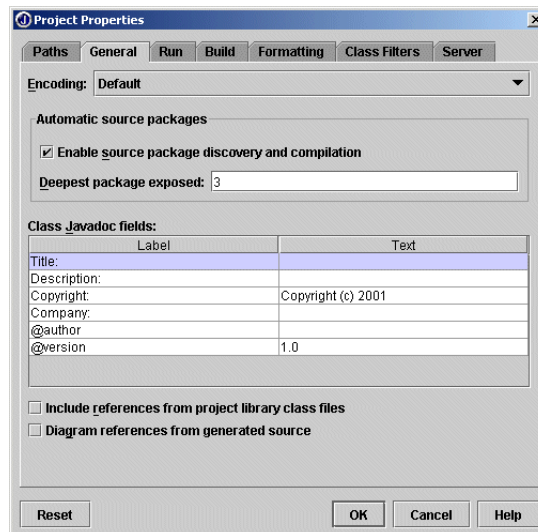
**Note** Checking this option is not required; it may slow down compiles and the refactoring process. However, if this option is off, JBuilder can't discover all references for the Find References command (see [“Finding references to a symbol” on page 12-8](#) for more information).

Additionally, your project must be up-to-date; that is, the timestamp on the class files and the source files must match. To make sure your project is up-to-date, compile it using the Project | Make Project command.

To set up JBuilder for references discovery and refactoring,

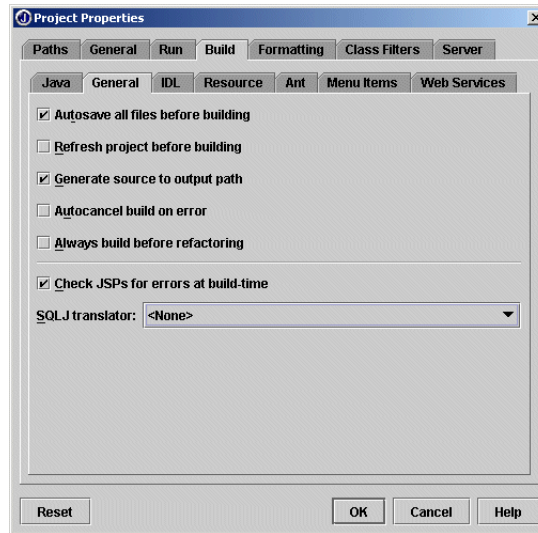
- 1 Choose Project | Project Properties to open the Project Properties dialog box.
- 2 Choose the General tab. Check the Include References From Project Library Class Files option.

The General tab of the Project Properties dialog box looks like this:



- 3 To always build your project before refactoring, choose the Build tab of the Project Properties dialog box. Then, click the General tab and choose the Always Build Before Refactoring option. This option is off by default. If you select this option, the refactoring is slower, but all cases are caught. If you leave this option off, the refactoring is faster, but there may be specific items missed in the refactoring.

The General tab of the Project Properties Build page looks like this:



- 4 Click OK to close the dialog box.
- 5 Choose File | Save All or click the Save All button on the toolbar.
- 6 Choose Project | Make Project to compile the entire project and load all references.



## Learning about a symbol before refactoring

Before you refactor, JBuilder provides several ways that you can learn about a symbol. You can find its definition. You can also find all references to the symbol; that is, all source files that use the symbol.

### Finding a symbol's definition

You can use Search | Find Definition or the Find Definition context menu command to determine where a symbol is defined. To find a symbol's definition,

- 1 Compile the project.
- 2 Select the symbol in the editor.

**3** Right-click the symbol and choose Find Definition.

The source file where the symbol is defined is opened in the editor.

- If the symbol is an instance of a class, the cursor is moved to the instance definition.
- If the symbol is a method, the class that defines the method is opened in the editor, with the cursor placed at the start of the method signature.
- If the symbol is a variable, and the variable is defined in the open class, the cursor moves to the variable definition. If the variable is `public` and defined in another class, the class is opened in the editor with the cursor placed on the definition.

**Important** In order for a definition to be located, you must have already compiled your project. The class that includes the definition must be on the `import` path or in the same package as the symbol.

## Finding references to a symbol

Before refactoring, you might also want to find all source files using a selected symbol. To locate all references to a symbol,












- 1 Compile the project.
- 2 Select the symbol in the editor or the structure pane.
- 3 Right-click the symbol and choose Find References or choose Search | Find References.

References are displayed on the Search Results tab of the message pane in order of discovery. Class and method references are sorted by category. Field and local variable references are sorted by file name. You cannot find references for a package or a property.

**Note** If you try to discover information about a symbol and JBuilder does not open a source file in the editor or display the Search Results tab, your project might not be compiled or might not be compiled with references from project libraries. For more information, see [“Setting up for references discovery and refactoring” on page 12-6.](#)

The following table details, by code symbol, the reference categories that can be displayed in the Search Results tab.

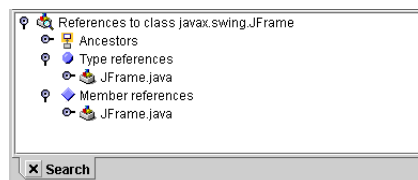
**Table 12.2** Find References details

Code symbol	Reference category
Class, inner class, or interface	 Ancestors- Classes that this class directly inherits from.
	 Descendents- Classes that directly descend from this class.
	 Type references- Classes that declare or instantiate the type of object for the class.
	 Descendents type references- Classes that are descendents or use descendents of the type of object for the class.
	 Member references- Members in this class.
	 Descendents member references- Members in classes that descend from this class.
Method or constructor	 Declarations- Locations where this method is declared.
	 Direct usages- Locations in directly instantiated classes that call this method.
	 Indirect usages- Locations in ancestor and descendent classes that indirectly call this method through an ancestor or descendent.
Field and local variable	 Writes- Locations where the field or local variable is written.
	 Reads- Locations where the field or local variable is read.

## Class references

If you have located references for a class, double-click a reference category in the Search Results tab to expand it. The source files where the class is referred to are listed. Click a source file, then click the reference to go directly to the reference in the editor.

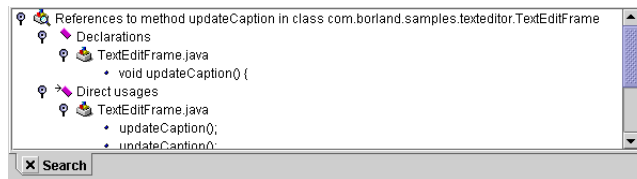
**Figure 12.1** Class references in the Search Results tab



## Method references

If you have located references for a method, double-click a reference category to expand it. The source files where the method is referred to are listed. Click a source file, then click the reference to go directly to the reference in the editor.

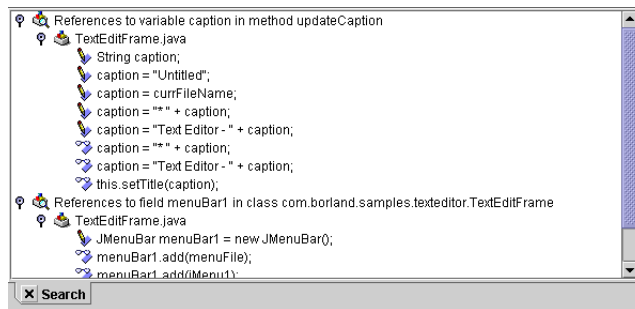
**Figure 12.2** Method references in the Search Results tab



## Field and local variable references

If you have located references for a field or local variable, double-click a file name to display the writes and reads for that symbol. Click the write or read reference to position the cursor on the reference in the editor.

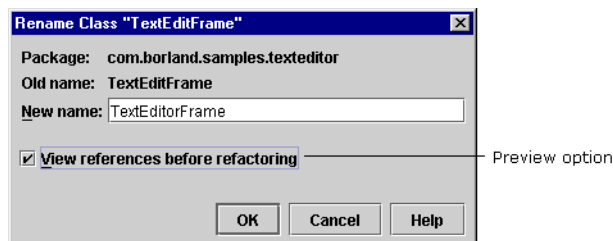
**Figure 12.3** Field and local variable references in the Search Results tab



## Viewing changes before a refactoring

For some types of refactoring, JBuilder provides the opportunity to view potential changes before committing the refactoring. You might want to preview changes when you are first using the refactoring tools, in order to carefully examine what JBuilder will change. The preview option, View References Before Refactoring, is shown below.

**Figure 12.4** Rename Class dialog box

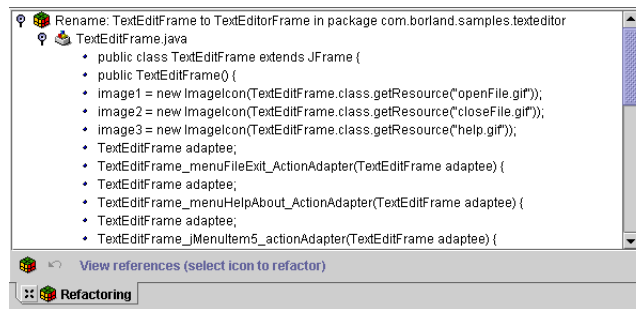


After you choose the preview option and click OK on the dialog box, potential changes are displayed on the Refactoring tab of the message pane. Potential changes, the lines that will change if you complete the refactoring, are displayed by file name, sorted in the order of discovery. To go directly to a reference in a source file, expand the file node and click the reference.

**Note** Some refactorings do not provide a preview option.

Before refactoring, the Refactoring tab will look similar to the following figure.

**Figure 12.5** Refactoring tab before refactoring



The Refactoring tab contains a Refactor button on the toolbar. It also displays an open cross-hatched X to show that the refactoring is unfinished. To finish the refactoring and commit the changes, click the Refactor button. The status bar in the Refactoring tab displays a message informing you of the progress.

**Note** If you edit any of the selected files before completing the refactoring, JBuilder won't allow the refactoring, since the files will be out-of-date. The Refactoring tab status bar displays the following message: "Files have changed - can't refactor."

The following table details the type of information displayed for a refactoring.

**Table 12.3** Refactoring details

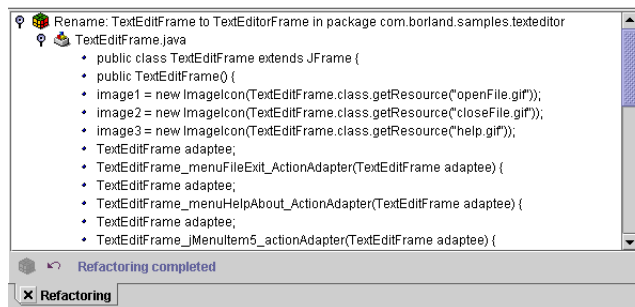
Code symbol	Type of refactoring	Information displayed
Package	Rename	Source files that contain a class reference that will change.
Class, inner class, or interface	Rename	Line locations in the current source file where the class is declared; includes constructors. Also lists source code locations where the class is used.

**Table 12.3** Refactoring details (continued)

Code symbol	Type of refactoring	Information displayed
Class	Move	Source code locations where the class' current package is declared or imported. Indicates if a package in the list of imports is added or deleted. (An <code>import</code> statement is added for any dependencies the class has on the package it is being moved from.)
Method	Rename	Source code locations where the method is declared and used. Indicates if a forwarding method is created.
Method	Change Parameters	Source code locations where the method is declared and called.
Field and local variable	Rename	Source code locations where the symbol is declared and called.
Property	Rename	Source code locations where the property is declared and where accompanying getter and setter are declared and called.

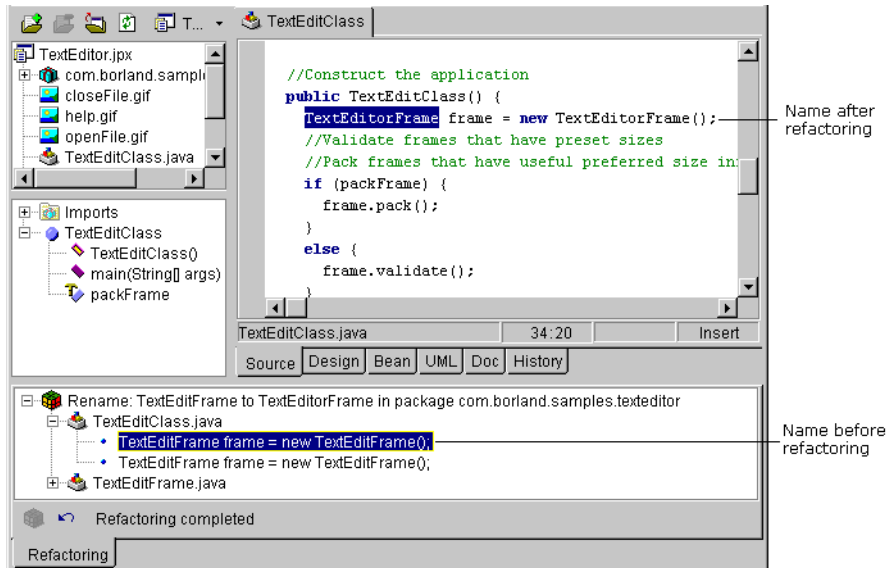
When the refactoring is completed, the status bar displays the message "Refactoring completed." The Refactor button is dimmed.

After refactoring, the Refactoring tab will look similar to the following figure.

**Figure 12.6** Refactoring tab after refactoring

After refactoring, the Refactor button is removed from the tab and the cross-hatch symbol closes to an X. The contents of the Refactoring tab, however, do not change. The original lines of source code are still displayed, so that you can compare the changes made by the refactoring. Click on an original source code line to go to the line that was changed.



**Figure 12.7** Source file and Refactoring tab after refactoring

## Executing a refactoring

To complete a refactoring in JBuilder, you first select the symbol or block of code you want to refactor. Then, right-click or use the Edit menu to choose the type of refactoring you want to complete. For most refactorings, JBuilder provides a dialog box where you can enter a new name and choose whether or not to preview the refactoring. For some refactorings, such as surrounding a block of code with `try/catch` statement, JBuilder will automatically complete the refactoring for you.

See the following topics for more information:

- [“Optimizing imports” on page 12-14](#)
- [“Rename refactoring a package” on page 12-17](#)
- [“Rename refactoring a class” on page 12-17](#)
- [“Move refactoring a class” on page 12-18](#)
- [“Rename refactoring a method” on page 12-19](#)
- [“Rename refactoring a local variable” on page 12-20](#)
- [“Rename refactoring a field” on page 12-21](#)
- [“Rename refactoring a property” on page 12-22](#)
- [“Changing method parameters” on page 12-22](#)

- “Extracting a method” on page 12-24
- “Introducing a variable” on page 12-25
- “Surrounding a block with try/catch” on page 12-26

## Optimizing imports

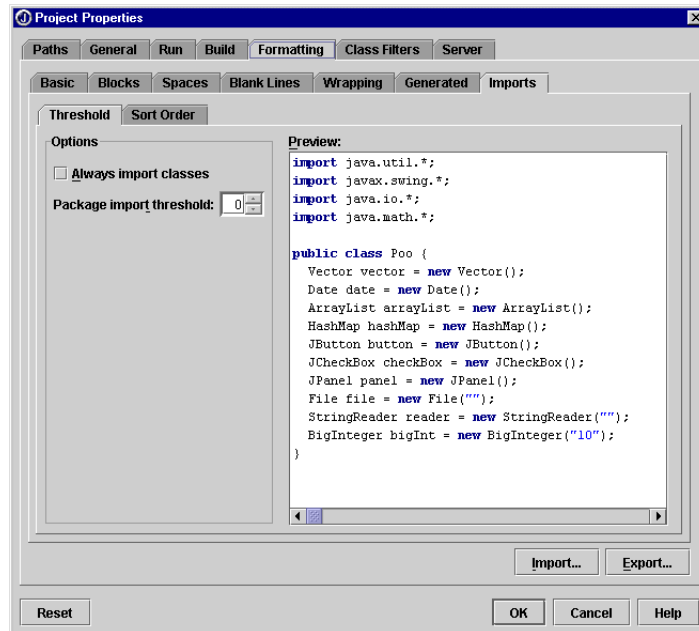
Use Optimize Imports to rewrite and reorganize your `import` statements according to the custom settings in the project properties. Optimize Imports also removes any `import` statements that are no longer used. You can customize the order of imports on the Imports tab of the Project Properties dialog box Formatting page (Project | Project Properties). Optimize Imports is available from the editor.

To customize the package import style, open the Project Properties dialog box. Choose one of the following:

- Right-click the project file in the project pane and choose Properties, or
- Select Project | Project Properties.

To set threshold and sort order options for imports,

- 1 Select the Formatting tab, then the Imports page. Choose the Threshold tab to set the package import threshold. The Threshold tab looks like this:

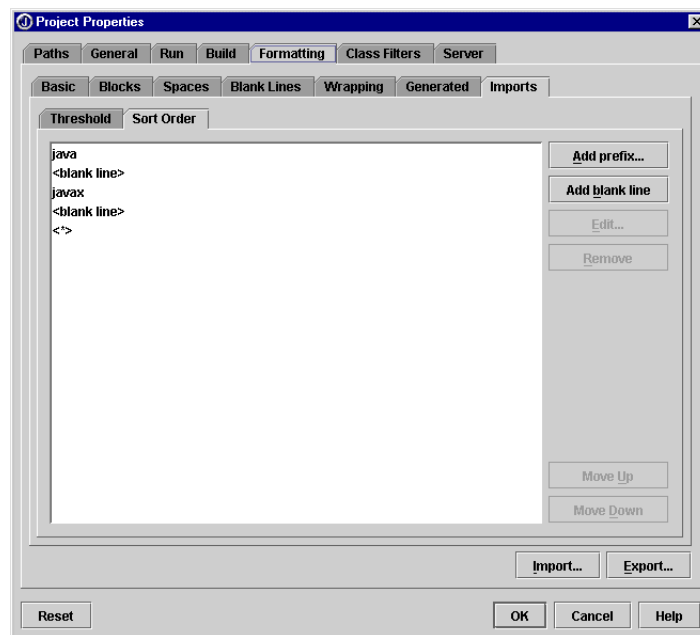


- 2 The Always Import Classes option determines if package import statements are added to your code. Check this option if you do not want to add package import statements to your code. Instead, individual classes are imported directory. When you use this option, the Package Import Threshold setting is ignored.
- 3 The Package Import Threshold sets how many classes must be imported from a package before rewriting class imports into a package import statement.

Classes up to this threshold are imported using individual class import statements. When the threshold is exceeded, the entire package is imported. For example, when three is entered in this field, and you use four or more classes from a package, the entire package will be imported.

The Preview box displays the results of different import thresholds settings.

- 4 Choose the Sort Order page to determine how imports are sorted. The Sort Order page looks like this:



- 5 To add an import that starts with a specified prefix, choose the Add Prefix button. Enter the prefix into the Add Prefix dialog box.

- 6 To insert an extra line break between import statements or groups of import statements, select the package you want to insert an extra line break below and click the Add Blank Line button.
- 7 To change a package import statement, select it and click the Edit button.
- 8 To remove an import from the list, select it and click Remove.
- 9 To move an import or a blank line within the list, click Move Up or Move Down.
- 10 To import a pre-defined formatting, click the Import button. The Import Code Formatting Settings dialog box is displayed. Choose the formatting you want to import and click OK.
- 11 To export and save the formatting, click the Export button. The Export Code Formatting Settings dialog box is displayed. Enter the name of the file you want to save the settings to and click OK. The file type must be `.codestyle`. Formatting setting files are saved to the `<.jbuilder>` directory.

**Note** Comments in the import section of the code are preserved.

**Tip** If you want to customize the order of import statements for *all* new projects, choose Project | Default Project Properties and make your modifications on the Import Style page in the Default Project Properties dialog box.

## Using Optimize Imports

To optimize imports,

- 1 Choose Project | Make Project to compile your project.
- 2 Choose Edit | Optimize Imports or right-click in the editor and choose Optimize Imports. You can also use the shortcut `Ctrl + I`. Additionally, you can choose the a symbol in the structure pane, right-click, and choose Optimize Imports.

**Tip** Choose Edit | Undo to undo Optimize Imports.

To optimize imports from the project pane,

- 1 Right-click the package in the project pane.
- 2 Choose Format Package.
- 3 Check the Optimize Imports option in the Format Code dialog box and click OK.

## Rename refactoring a package

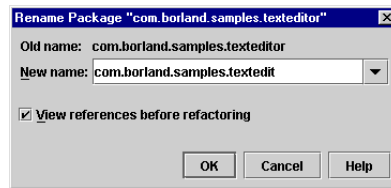
---

You can rename refactor a package from the editor, the structure pane, or a UML class or package diagram. Rename refactoring a package renames the package and the entire sub-tree of packages to the new root package name. It also moves the package and all class names to the new name and source directory. The existing source directory structure for that package is deleted.

To rename refactor a package,

- 1 Right-click the package name in a UML class or package diagram, the structure pane, or in the editor.
- 2 Choose Rename.

The Rename Package “package name” dialog box is displayed.



- 3 Enter the new name for the package in the New Name field.
- 4 Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button on the toolbar to complete the refactoring.)



The package rename refactoring is prevented if the new package name already exists or is invalid.

## Rename refactoring a class

---

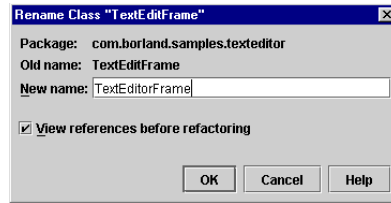
You can rename refactor a class, inner class, or interface from either the editor, the structure pane, or the UML class diagram. Rename refactor for an outer public class renames all declarations of and all usages of the class and the source file. If you select a constructor, the rename refactoring renames the class.

To rename refactor a class, inner class, or interface,

- 1 Open the class file you want to rename in the editor or as a UML diagram.
- 2 In the editor, UML diagram, or structure pane, right-click the class, inner class, or interface you want to change the name of.

**3** Choose Rename.

The Rename Class “ClassName” dialog box is displayed.



**4** Enter the new name of the class in the New Name field.

- 5** Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button on the toolbar to complete the refactoring.)



The refactoring is prevented if the class identifier is invalid. If the class is not the outer public class and there is another non-outer public class of the desired new name, the class isn't renamed.

## Move refactoring a class

---

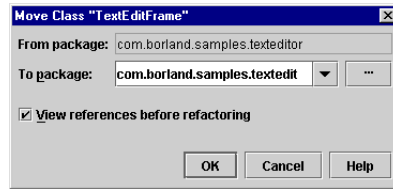
You can move a class to a new package from the editor, the structure pane, or the UML class diagram. Move refactoring a class moves that class to a new package as long as the new package does not already contain a source file of the new name. The package and import statements in the class source file, as well as in all classes that reference the moved class, are updated. (An `import` statement is added for any dependencies the class has on the package it is being moved from.) The class must be the top level public class.

To move a class to a new package,

- 1** Open the class file you want to move in the editor or as a UML diagram.
- 2** In the editor, UML diagram, or structure pane, right-click the class name.

### 3 Choose Move.

The Move Class “ClassName” dialog box is displayed.



- 4 Enter the name of the package the class is being moved to in the To Package field.
- 5 Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button to complete the refactoring.)



The class isn't moved if the class identifier is invalid or if the source file name already exists in the new package. JBuilder adds an `import` statement for the old package name, if needed.

**Note** If you move a class to a package that doesn't exist, JBuilder creates the new package, adds it to your project, creates the new source directory, and moves the class to it. It also updates package names and import statements. Additionally, if the package no longer contains any classes, JBuilder removes that package from the project and deletes its source directory.

## Rename refactoring a method

You can rename refactor a method from either the editor, the structure pane, or a UML diagram. Rename refactoring a method renames the method, all declarations of that method, and all usages of that method. The method can be renamed from the selected class down in the hierarchy or in the entire hierarchy. A forwarding method, that passes on the method call to the new method, can be created. This allows your public API to remain intact.

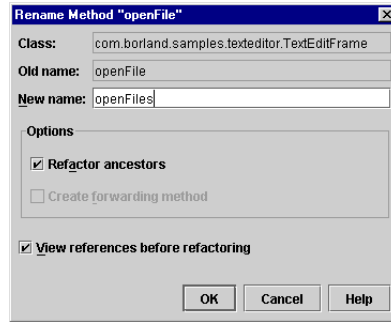
**Note** Renaming a method does not rename overloaded methods; that is, methods with the same name but with different method signatures.

To rename refactor a method,

- 1 Open the source file containing the method you want to rename in the editor or as a UML class diagram.
- 2 In the editor, UML diagram, or structure pane, right-click the method you want to change the name of.

### 3 Choose Rename.

The Rename Method “methodName” dialog box is displayed.



### 4 Enter the new name of the method in the New Name field.

### 5 The Refactor Ancestors option (on by default) renames methods in classes that this class inherits from.

### 6 Turn off Refactor Ancestors to rename the method only in this class and in its descendents. You can then choose to add a forwarding method by clicking the Create Forwarding Method option.

### 7 Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button on the toolbar to complete the refactoring.)



The refactoring is prevented if the new method name already exists in the file where it is declared. If the name exists in other files in the direct inheritance, a warning is issued. If you are refactoring with Refactor Ancestors, a warning can also be displayed if the method exists, but is not in the editable source path. For example, if the method exists in a library, you won't be able to refactor it, as libraries are read-only.

## Rename refactoring a local variable

You can rename refactor a local variable only from the editor. A local variable rename refactoring changes the declaration and usages of that variable to the new name. Note that a method parameter is treated as a local variable.

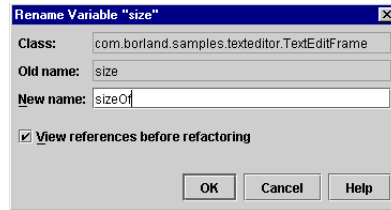
To rename refactor a local variable,

### 1 Right-click the local variable you want to change the name of.



## 2 Choose Rename.

The Rename Variable “variableName” dialog box is displayed.



## 3 Enter the new name of the variable in the New Name field.

## 4 Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button on the toolbar to complete the refactoring.)



The refactoring is prevented if the new name exists in the class that declared the original variable.

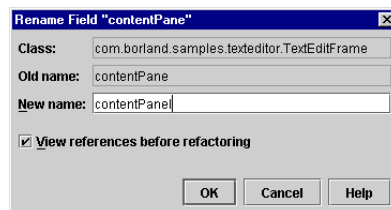
## Rename refactoring a field

You can rename refactor a field from either the editor, the structure pane, or a UML class diagram. A field rename refactoring changes the declarations and usages of that field to the new name.

To rename refactor a field,

- 1 Open the source file containing the field you want to rename in the editor or as a UML class diagram.
- 2 In the editor, UML diagram, or structure pane, right-click the field you want to change the name of.
- 3 Choose Rename.

The Rename Field “variableName” dialog box is displayed.



## 4 Enter the field’s new name in the New Name field.

## 5 Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button on the toolbar to complete the refactoring.)



The refactoring is prevented if the new name exists in the class that declared the field. If there are scope conflicts between the new name and the old name, the `this` keyword is added to the new field name. A warning is displayed if the new name overrides or is overridden by an existing field in a superclass or subclass.

## Rename refactoring a property

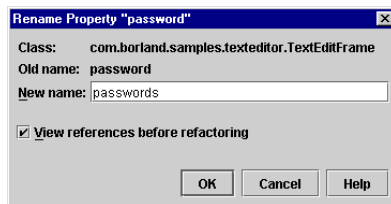
---

This is a feature of  
JBuilder Enterprise

You can rename refactor a property only from a UML class diagram. A property rename refactoring changes all declarations of that property, as well as its getter and setter methods.

To rename refactor a property,

- 1 Right-click the property you want to change the name of.
- 2 Choose Rename. The Rename Property “propertyName” dialog box is displayed.



- 3 Enter the new name of the property in the New Name field.
- 4 Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button on the toolbar to complete the refactoring.)



The refactoring is prevented if the new name exists in the class that declared the original property.

## Changing method parameters

---

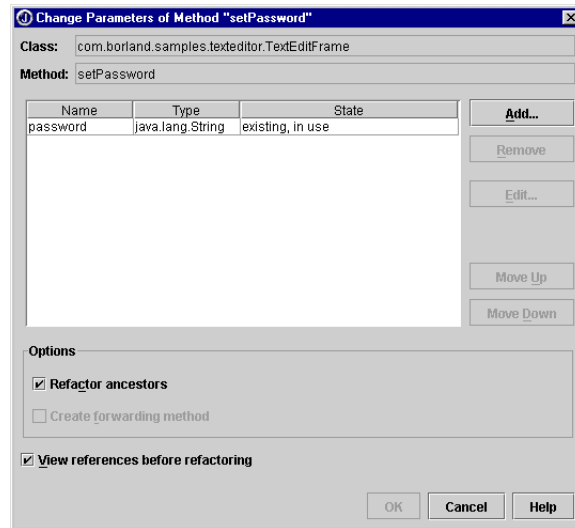
You can add, delete and re-order a method’s parameters from the editor, the structure pane or from a UML diagram. You can edit a newly added parameter before you close the Change Parameters dialog box; however, you cannot edit an existing parameter.

To change a method’s parameters,

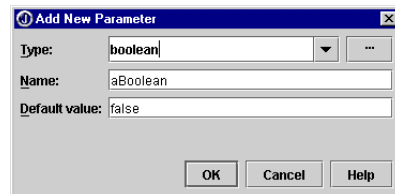
- 1 Right-click the signature of the method you want to change parameters for.

## 2 Choose Change Parameters of “methodName”.

The Change Parameters dialog box is displayed. Existing parameters are displayed in the list.



- The Name column displays the name of the parameter.
  - The Type column displays the Java type.
  - The State column shows if the parameter is new or existing and if it is in use in your code. For new parameters, it shows the default value.
- 3 To add a new parameter, click the Add button. The Add New Parameter dialog box is displayed where you choose the parameter’s type, enter a name for the parameter, and assign a default value.



- 4 To edit a newly added parameter, select the parameter and click Edit. The Edit New Parameter dialog box is displayed where you can change the name, type, or default value.
- 5 To remove a newly added parameter, choose the parameter and click the Remove button. You cannot remove existing, in use parameters.
- 6 To rearrange the order of the method parameters, use the Move Up and Move Down buttons.

- 7 The Refactor Ancestors option (on by default) refactors methods in classes that this class inherits from. Turn off Refactor Ancestors to refactor the method only in this class and in its descendents. You can then choose to add a forwarding method by clicking the Create Forwarding Method option.
- 8 Click the View References Before Refactoring option if you want to view the changes before completing the refactoring. Otherwise, click OK to complete the refactoring. (If you preview, click the Refactor button on the toolbar to complete the refactoring.)



The refactoring is prevented if the new method signature already exists in the file where it is declared. If the signature exists in other files in the direct inheritance, a warning is issued. If you are refactoring with Refactor Ancestors, a warning can also be displayed if the same method exists, but is not in the editable source path. For example, if the method exists in a library, you won't be able to refactor it, as libraries are read-only.

Note that the refactoring is prevented if the new parameter name or type is not a valid Java identifier.

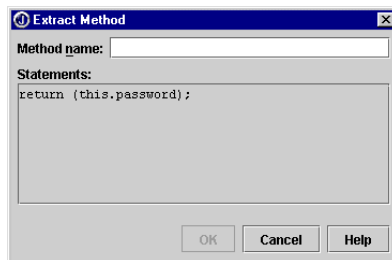
## Extracting a method

---

The Extract Method refactoring allows you to turn a selected code fragment into a method. You can access this refactoring from the editor.

To extract a method,

- 1 In the editor, select the block of code you want to turn into a method.
- 2 Right-click and choose Extract Method. The Extract Method dialog box is displayed.



- 3 Enter a new name for the method in the New Method Name field. The name should explain the purpose of the method.
- 4 Click OK to complete the refactoring.
- 5 Use Edit | Undo to undo the refactoring.

JBuilder moves the extracted code outside of the current method, determines the needed parameter(s), generates local variables if necessary, and determines the return type. It inserts a call to the new

method in the code where the code fragment resided. JBuilder will not allow the refactoring if more than one variable is written to or if it is read after the block.

**Note** If the set of statements is not completely selected, JBuilder will attempt to expand the selection out to the nearest enclosing expression or statement.

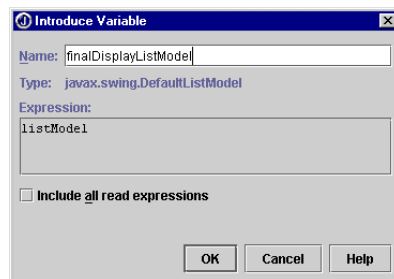
## Introducing a variable

---

Use the Introduce Variable refactoring to replace the result of a complex expression, or part of the expression, with a temporary variable name. The name should explain the purpose of the expression or sub-expression. This is also known as an explaining variable.

To introduce a variable,

- 1 Select the complex expression you want to replace with a temporary variable.
- 2 Right-click and choose Introduce Variable. The Introduce variable dialog box is displayed.



- 3 Enter the name of the new variable in the Variable Name field.
- 4 Check the Include All Read Expressions option to replace all reads from that expression. If this option is off, only the current selection is replaced.
- 5 Click OK to close the dialog box. The refactoring is automatically completed.
- 6 Use Edit | Undo to undo the refactoring.

A final temporary variable with the selected variable name is generated and initialized in the correct place. The original expression is replaced with the newly generated variable.

## Surrounding a block with try/catch

---

You can place a `try/catch` statement around a selected block of code. JBuilder will detect all checked exceptions in a block and adds specific blocks for each checked exception. This refactoring is available from the editor.

To surround a block with a `try/catch` statement,

- 1 Select the block of code in the editor.
- 2 Right-click and choose Surround With Try/Catch.

The code is surrounded with a `try/catch` statement. If the selected block is not a valid block of statements, an error will display in the Refactoring tab and the refactoring will be prevented. Use `Edit | Undo` to undo the refactoring.

## Undoing a refactoring

---



Once you've completed a refactoring, you can reverse it by clicking the Undo button on the Refactoring toolbar. Undo immediately, before you make other changes to files. All changes are reversed. You can then redo the refactoring by clicking the Refactor button on the toolbar. Undo is active as long as the Refactoring tab is open.

When you use a refactoring that does not display output in the Refactoring tab, you can undo changes with `Edit | Undo`. Refactorings that do not display output are:

- Optimize Imports
- Extract Method
- Introduce Variable
- Surround With Try/Catch

## Saving refactorings

---

After you complete a refactoring, you should immediately save the files in your project with the `File | Save All` command. If you are using a version control system, commit or check in the changes right away.

If you try to close your project before saving files, JBuilder displays the Save Modified Files dialog box where you can select the files you want to save. If you don't save files, your source code reverts to its state before the refactoring(s).

**Important** Refactoring works on files that may not be open in the editor at the time of the refactoring. JBuilder automatically saves changes to those files. JBuilder makes these changes and saves files so that your source code isn't in an inconsistent state.

## Unit testing

Unit testing is a feature of  
JBuilder Enterprise.

Unit testing means writing tests for small, discretely defined portions of your code, such as a method, and then running and analyzing the tests. When a developer does unit testing as part of their development process, it means writing many small repeatable tests and running them on a regular basis. The benefits are better confidence in the quality of the code and early discovery of *regressions* when code modifications are made. A regression is a bug that has been introduced in code that was previously working. Many methodologies recommend running unit tests as part of the build process every time you build your project. If the unit tests don't pass, these methodologies consider the build process to have failed.

### JUnit

---

JUnit is an open source framework for unit testing written by Erich Gamma and Kent Beck. JUnit provides a variety of features which support unit testing, among them two classes, `junit.framework.TestCase` and `junit.framework.TestSuite`, which are used as base classes for writing unit tests. JUnit also provides three different kinds of test runners, TextUI, SwingUI, and AwtUI. Of these three test runners, two of them, TextUI and SwingUI, are available within the JBuilder IDE. For more information about JUnit, visit <http://www.junit.org>. JUnit documentation is also available in your `<jbuilder>/thirdparty/<junit>/doc` directory.

JBuilder integrates JUnit's unit testing framework into its environment. This means that you can create and run JUnit tests within the JBuilder IDE. In addition to JUnit's powerful unit testing features, JBuilder adds wizards for creating test cases, test suites, and test fixtures, and a test runner called JBuilderTestRunner which combines both text and GUI elements in its output and integrates seamlessly into the JBuilder IDE.

**See also**

- [“Discovering tests” on page 13-3](#)
- [“Creating JUnit test cases and test suites” on page 13-4](#)
- [“Running tests” on page 13-13](#)

## Cactus

---

Cactus extends JUnit to provide unit testing of server-side Java code. It does this by redirecting your test case to a server-side proxy. For more information about Cactus, visit <http://jakarta.apache.org/cactus/index.html>. Cactus documentation is also available in your `<jbuilder>/thirdparty/<jakarta-cactus>/doc` directory.

JBuilder provides several features that make Cactus testing easier. The Cactus Setup wizard allows you to configure your project for Cactus test support. The EJB Test Client wizard can generate a Cactus test case for your Enterprise JavaBean (EJB). JBuilder’s integration of Cactus into its environment means that you can run your Cactus tests within the JBuilder IDE when a properly configured server and project are available.

**See also**

- [“Working with Cactus” on page 13-11](#)
- [“Cactus Setup wizard” on page 13-11](#)
- “Running and testing an enterprise bean” in *Enterprise JavaBeans Developer’s Guide*

## Unit testing features in JBuilder

---

JBuilder’s unit testing features integrate JUnit and Cactus into JBuilder’s IDE and provide tools for writing unit tests and organizing them into test suites, running tests, analyzing tests, and debugging tests. JBuilder provides a set of predefined test fixtures for performing common tasks that your tests may require. JBuilder’s JBuilderTestRunner offers a way to run tests that combines both text and GUI output. JBuilder includes the following unit testing features:

- Test Case wizard
- Test Suite wizard
- EJB Test Client wizard
- JDBC Fixture



- JNDI Fixture
- Testing by comparison
- Custom Fixture wizard
- Cactus testing
- Test running
- JUnitRunner
- JUnit TextUI support
- JUnit SwingUI support
- Test stack trace filter
- Test debugging
- JUnit Test Collector

## Discovering tests

---

By default, JBuilder automatically identifies a class as a test case if it extends `junit.framework.TestCase` or `junit.framework.TestSuite`. If a class is identified as a test case and an appropriate runtime configuration exists, right-clicking either the name of the source file in the project pane or the tab containing the name of the source file when it's open in the editor brings up a context menu which contains Run Test and Debug Test options. An Optimize Test option is also available when Borland Optimizeit is properly installed.

An alternate method of identifying tests is provided by JUnit Test Collector.

### JUnit Test Collector

---

JUnit Test Collector is a feature of JBuilder that provides a graphical user interface (GUI) for the `PackageTestSuite` class. This is useful for test discovery, so that you don't need to maintain a list of all your test classes. JUnit Test Collector is available in the Runtime Configuration Properties dialog box for a Test type runtime configuration. You switch it on by selecting the Package radiobutton on the Run page of this dialog box.

To add a Test type run configuration that uses JUnit Test Collector,

- 1 Select Project | Project Properties.
- 2 Select the Run page of the Project Properties dialog box.
- 3 Click the New button.

- 4 Select the Run page of the Runtime Configuration Properties dialog box.
- 5 Set the Type to Test.
- 6 Click the Package radiobutton. This enables JUnit Test Collector and disables the default test discovery mode.
- 7 Specify whether or not you want the test scanner to include sub-packages by checking or unchecking Include Sub-packages.
- 8 Specify strings that your test class names start or end with in the Name Starts With and Name Ends With fields. Doing so restricts the tests that the test scanner finds to only those that contain these strings. This step is optional.
- 9 Click OK to save the runtime configuration and close the Runtime Configuration Properties dialog box.
- 10 Click OK to close the Project Properties dialog box.

Tests which match the filter you provided in the Runtime Configuration Properties dialog box will now be correctly identified as tests. This means that the Run Test and Debug Test options will be displayed on the context menu when you right-click one of the matching tests in the project pane. An Optimize Test option will also be available if Borland Optimizeit is properly installed.

#### See also

- [“Setting runtime configurations” on page 7-6](#)

## Creating JUnit test cases and test suites

---

A test case is an instance of `junit.framework.TestCase`. A test case class contains one or more methods that exercise one or more parts of a class in the application under test. It also contains `setUp()` and `tearDown()` methods. The `setUp()` method is used to do any required setup that needs to be done before each test method is run. The `tearDown()` method is used to clean up and release resources after each test method has been run. When JUnit tests are run, a new instance of the test case class is created for each test method. The `setUp()` and `tearDown()` methods are run once for each instance. For example, given a test case called `MyTestCase` which contains the methods `testMethod1()` and `testMethod2()`, the order of execution would be:

- 1 Two instances of `MyTestCase` are created by the test runner.
- 2 The `setUp()` method is called.
- 3 The `testMethod1()` method is called.

- 4 The `tearDown()` method is called.
- 5 The `setUp()` method is called.
- 6 The `testMethod2()` method is called.
- 7 The `tearDown()` method is called.

A test case and a test suite both extend `TestCase`. The difference between a test case and a test suite is that a test case contains individual test methods, while a test suite is used to organize test cases into a logical group and run them as a group. A test suite can call any number of test cases or other test suites.

An important goal of unit testing is to create repeatable tests. If tests are repeatable they always give the same result when the software under test is working properly. If a tested method is no longer working as expected, the test will fail. If you run unit tests each time you make modifications to your software, it helps ensure that you have not introduced any errors or regressions.

The number of tests you write is your decision. Some developers have a policy of writing tests for every public method in their code. You don't have to achieve this level of coverage to provide some protection against regressions. You might want to concentrate at first on just writing tests for the areas of the software that are the most critical, or those that are the most likely to break.

Test methods should return an expected result if the test passes. If the test fails, they should return information which is useful in determining the cause of the failure. The Test Case wizard creates skeletons of test methods and you make the decision about what sort of results are meaningful and provide the implementation.

### See also

- [Chapter 21, "Tutorial: Creating and running test cases and test suites"](#)

## The Test Case wizard

---

The Test Case wizard is used to create test classes that extend `TestCase` and contain skeleton methods for exercising the methods of the class under test. To go to the Test Case wizard, select `File | New` from the menu, click the `Test` tab of the object gallery, select `Test Case`, and click `OK`. The Test Case wizard creates new test cases in the test source directory as specified on the `Paths` page of the `Project Properties` dialog box. To view or edit the test source directory, go to `Project | Project Properties`, select the `Paths` page, and click the `Source` tab.

The Test Case wizard lets you select the class and the methods to test, make use of predefined test fixtures, and create a new runtime

configuration for the test case. For more specific information on the UI of the Test Case wizard, click the Help button in the wizard.

### See also

- [Chapter 21, “Tutorial: Creating and running test cases and test suites”](#)
- [“Using predefined test fixtures” on page 13-8](#)

## Adding test code to your test cases

---

The Test Case wizard creates the skeletons of test cases, but it’s up to you to fill in the actual test code. The Test Case wizard flags the areas of code that need to be completed with `@todo` Javadoc comments. These comments are shown in the structure pane in the To Do node of the tree. To complete your test cases, you will have to add test code to each test method. You will often also need to provide non-null values for certain variables which will also be flagged with `@todo` comments by the Test Case wizard.

Here is an example of a simple test method:

```
public void testSum() {
    assertEquals( 2, sum(1,1) );
}
```

Test methods must be `public`, they must be `void`, and they must take no arguments.

When you add test code to your test methods, you need a way to determine whether a test passed or failed. You can design a test method to test for various conditions and report failure if they are not met. The most common way of doing this is to call one of the assert methods in `junit.framework.Assert`, for example:

- `assertEquals()`—makes the assertion that the arguments passed to it are equal.
- `assertTrue()`—makes the assertion that a boolean expression passed to it evaluates to true.
- `assertNotNull()`—makes the assertion that an argument passed to it is not null.

There are several overloaded versions of these methods in `junit.framework.Assert`. Their various method signatures take different types of arguments, making them more flexible. Any of these methods trigger a test failure, which is reported by a test runner, if the condition it tests is not met. If a test method finishes without triggering a failure, the test runner reports success. You can call any of the assert methods directly from your test case because `TestCase` is a subclass of `Assert`.

**Tip** To view the various methods in `junit.framework.Assert`, open a test case in the editor, double-click the parent class, `TestCase`, in the structure pane, then double-click its parent class, `Assert`.

You can also write a test method that throws an exception. Here is an example:

```
public void testException() throws Exception {
    throw new Exception("ouch!");
}
```

When a test throws an exception that it does not handle, the test runner will report a failure for this method.

For more information on writing tests with JUnit, see the article by Kent Beck and Erich Gamma, “JUnit Test Infected: Programmers Love Writing Tests” on the JUnit web site.

## The Test Suite wizard

---

The Test Suite wizard is used to create a test suite that groups test cases so they can be run as a batch. To go to the Test Suite wizard, select **File | New** from the menu, click the **Test** tab of the object gallery, and select **Test Suite**.

The Test Suite wizard lets you select the test cases to be included in the test suite and create a runtime configuration. For more specific information on the UI of the Test Suite wizard, click the **Help** button in the wizard.

### See also

- [Chapter 21, “Tutorial: Creating and running test cases and test suites”](#)

## The EJB Test Client wizard

---

The EJB Test Client wizard, available on the **Enterprise** page of the object gallery, allows you to create three different types of test clients for testing your Enterprise JavaBeans (EJB). Of these three types of test clients, two of them, JUnit test client and Cactus JUnit test client, are designed for unit testing.

**Tip** Although it’s possible to create a test case that tests an EJB using the Test Case wizard, it’s better to use the EJB Test Client wizard when testing an EJB. That’s because the EJB Test Client wizard generates more EJB-specific code.

### See also

- “Running and testing an enterprise bean” in *Enterprise JavaBeans Developer’s Guide*

## Using predefined test fixtures

---

Test fixtures are utility classes that can be used by tests to perform routine tasks to create the desired test environment. One example of this is managing database connections to data that is used for testing purposes.

JBuilder’s Test Case wizard can automatically install test fixtures if they provide a constructor that takes an `Object` argument and contain `setUp()` and `tearDown()` methods. Here is a basic example of a valid fixture:

```
public class CustomFixture1 {

    public CustomFixture1(Object obj) {
        // code goes here
    }

    public void setUp() {
        // code goes here
    }

    public void tearDown() {
        // code goes here
    }

}
```

To install a fixture of this type in a new test case, select the fixture in step 3 of the Test Case wizard. The fixture will be instantiated by the test case, and its `setUp()` and `tearDown()` methods will be invoked.

JBuilder provides the following three predefined fixtures for performing common tasks:

- JDBC fixture
- JNDI fixture
- Comparison fixture

You can also create your own custom test fixtures using the Custom Fixture wizard.

### JDBC fixture

---

The JDBC fixture, `com.borland.jbuilder.unittest.JdbcFixture`, can be used by test cases for managing JDBC connections. Test methods within the test case can use the `getConnection()` method to get a JDBC connection. To

specify a JDBC connection, use the `setUrl()` and `setDriver()` methods. The `runSqlFile()` method is used for running SQL script files.

The easiest way to create a JDBC Fixture is by using the JDBC Fixture wizard. The JDBC Fixture wizard creates a class that extends `JdbcFixture`. By extending `JdbcFixture`, you can specify a JDBC connection to use and provide other functionality as needed. Here is a summary of some of the most commonly used methods in `JdbcFixture`:

- `dumpResultSet()` dumps the values in a result set to a `Writer`. Takes a `java.sql.ResultSet` and a `java.io.Writer` as parameters.
- `getConnection()` returns a `java.sql.Connection` object defining the JDBC connection.
- `runSqlBuffer()` runs a SQL statement contained in a `StringBuffer`.
- `runSqlFile()` reads a SQL script from a file and runs it. Takes a `String` indicating the location of the file and a `boolean` as parameters.
- `setDriver()` sets the `Driver` property of the JDBC connection. Takes a `String` as a parameter.
- `setUrl()` sets the `URL` property of the JDBC connection. Takes a `String` as a parameter.
- `setUsername()` sets the username for accessing the JDBC connection. Takes a `String` as a parameter.
- `setPassword()` sets the password for accessing the JDBC connection. Takes a `String` as a parameter.

**Tip** When you create a JDBC fixture using the wizard, it extends `JdbcFixture`. You can view the inherited class structure by double-clicking the parent class node in the structure pane when your JDBC Fixture is open in the editor or by right-clicking the name of the parent class in the editor and selecting **Find Definition** from the context menu.

For more specific information about the UI of the JDBC Fixture wizard, click the **Help** button in the wizard.

### See also

- [Chapter 22, “Tutorial: Working with test fixtures”](#)

## JNDI fixture

---

You can create a JNDI fixture, which is a class that facilitates performing JNDI lookups, using the JNDI Fixture wizard. The JNDI Fixture wizard is available on the **Test** page of the object gallery. For more specific information about the UI of the JNDI Fixture wizard, click the **Help** button in the wizard.

## Comparison fixture

---

A comparison fixture is used for recording output from a test run and then comparing output from subsequent test runs against previous test output. A comparison fixture is a class that extends `com.borland.jbuilder.unittest.TestRecorder`. The `TestRecorder` class extends `java.io.Writer`, so you can use your comparison fixture anywhere a `Writer` is required. You can generate a comparison fixture using the Comparison Fixture wizard, available from the Test page of the object gallery.

A `TestRecorder` contains four constants for setting the recording mode:

- **UPDATE**—The comparison fixture compares new output to an existing output file, or creates the output file if it does not exist and records output to it.
- **COMPARE**—The comparison fixture always compares new output to the output that already exists.
- **RECORD**—The comparison fixture records all output, overwriting any previous output existing in the output file.
- **OFF**—The comparison fixture is disabled.

Keep in mind that if you change a test case or test suite after test output has already been recorded by inserting or deleting new string output, you must reinitialize the data file. Use **RECORD** instead of **UPDATE** when your tests have changed, or delete the existing data file. The data file is a binary file located in the same directory as your test source files. It has the same name as your test case.

Here is a summary of some of the most commonly used methods of a comparison fixture:

- `print()` prints a string that is passed to it as a parameter.
- `println()` prints a string that is passed to it as a parameter with a line break.
- `compareObject()` invokes the `equals()` method of an object to compare an object passed to it to an object that was previously recorded using `recordObject()`.
- `recordObject()` records an object so that it can later be compared to another object using `compareObject()`.

For more specific information about the UI of the Comparison Fixture wizard, click the Help button in the wizard.

### See also

- [Chapter 22, “Tutorial: Working with test fixtures”](#)



## Creating a custom test fixture

---

You may want to write your own custom test fixtures to perform tasks that need to be done in many of your tests. Your tests can then share your custom fixture. The Custom Fixture wizard is useful for generating a skeleton for a test fixture or creating a wrapper for existing test fixture code. The custom fixture skeleton includes `setUp()` and `tearDown()` methods. You can find the Custom Fixture wizard on the Test page of the object gallery. For more specific information about the UI of the Custom Fixture wizard, click the Help button in the wizard.

## Working with Cactus

---

Cactus extends JUnit to provide unit testing of server-side Java code. It is useful in testing your Enterprise JavaBeans (EJB) and web applications. JBuilder provides features which make Cactus testing easier.

- Cactus Setup wizard—Configures your project to work with Cactus so that you can run Cactus tests in the JBuilder IDE.
- EJB Test Client wizard—Helps you to create a Cactus test client for your EJB.

The primary goal of JBuilder's Cactus support is to facilitate EJB testing with Cactus. Testing your EJB with Cactus is discussed in more detail in "Running and testing an enterprise bean" in *Enterprise JavaBeans Developer's Guide*.

You may also want to use Cactus to test other types of server-side Java code. Even if testing an EJB is not your goal, you can still use the Cactus Setup wizard to configure your project for Cactus testing and facilitate proper deployment of the required files.

### Cactus Setup wizard

---

The Cactus Setup wizard configures your project to use Cactus. This makes it possible to run Cactus tests within the JBuilder IDE. The wizard is available by selecting Wizards | Cactus Setup.

To configure your project for Cactus:

- 1 Select Wizards | Cactus Setup. The Cactus Setup wizard opens.
- 2 Select the WebApp to which the wizard will add Cactus test support. You may use the default WebApp, an existing WebApp, or click the New button to open the Web Application wizard and create a new WebApp.

- 3 Choose the logging settings for the Cactus logs. Specify the locations for the Cactus server and client logs, or uncheck Enable Logging if you don't want any logs.
- 4 Click Next.
- 5 Select the archives to deploy on the server and redeploy before each test. This keeps the archives in sync with the project. If any of the archives are shown in red with an exclamation point before the name, it means that the physical file does not yet exist. This is probably because the archive has not yet been built. It won't cause any problem to select one of these archives, as long as you remember to build the archive before attempting to run Cactus tests.
- 6 Select a Server runtime configuration. You may create a new one using the New button.
- 7 Select a Test runtime configuration. You may create a new one using the New button.
- 8 Click Finish. Your project is now configured for use with Cactus.

### See also

- “Configuring your project for testing an EJB with Cactus” in *Enterprise JavaBeans Developer's Guide*

## Creating a Cactus test case for your Enterprise JavaBean

---

You may want to use Cactus to test your Enterprise JavaBeans (EJB). JBuilder provides the EJB Test Client wizard, which can generate three different types of EJB test clients. One of these is a Cactus test client. To open the EJB Test Client wizard:

- 1 Select File | New.
- 2 Select the Enterprise page of the object gallery.
- 3 Select EJB Test Client and click OK.

**Note** The EJB Test Client wizard will not be available if your project is not properly configured to use a server that can support EJB services.

Testing your EJB is outside the scope of this chapter. This topic is covered by “Running and testing an enterprise bean” in *Enterprise JavaBeans Developer's Guide*.

### See also

- “Running and testing an enterprise bean” in *Enterprise JavaBeans Developer's Guide*
- “Configuring the target application server settings” in *Enterprise JavaBeans Developer's Guide*

## Running Cactus tests

---

Running Cactus tests is more complicated than running other types of unit tests, since you need to make sure you have a properly configured server, the correct deployment descriptors, and properly configured Test and Server runtime configurations. Apart from these configuration issues, the main difference between running a Cactus test and running any other JUnit test is that for a Cactus test, you first need to start the server.

Once the configuration is correct and the server is running, running Cactus tests within the JBuilder IDE is similar to running other JUnit tests. When your project is configured correctly, all you need to do to run Cactus tests is:

- 1 Start the server using the Server runtime configuration.
- 2 Right-click the Cactus test file in the project pane.
- 3 Select Run Test Using <test configuration> from the context menu. The test runs in the test runner that's specified in your Test runtime configuration.

The various issues involved in configuring your server and any required deployment descriptors are outside the scope of this chapter. They are discussed in more detail in *Enterprise JavaBeans Developer's Guide* and *Web Application Developer's Guide*.

### See also

- "Running and testing an enterprise bean" in *Enterprise JavaBeans Developer's Guide*
- "Configuring the target application server settings" in *Enterprise JavaBeans Developer's Guide*
- "Working with WebApps and WAR files" in *Web Application Developer's Guide*
- ["Running tests" on page 13-13](#)

## Running tests

---

Three different test runners are available for running your tests. JBuilder's default test runner is called JBuilderTestRunner. If you prefer, you can use JUnit's TextUI or SwingUI as your test runner. You specify the test runner you want to use in your Test type runtime configuration. To select a test runner:

- 1 Select Project | Project Properties from the menu.
- 2 Select the Run page.

- 3 Select an existing Test type runtime configuration and click Edit, or click New if there is no existing Test type runtime configuration. The Runtime Configuration Properties dialog box is displayed.
- 4 Enter a name for the runtime configuration, if needed.
- 5 Select the Run page of the Runtime Configuration Properties dialog box.
- 6 Set the runtime configuration Type to Test.
- 7 Select your preferred test runner from the Test Runner drop-down list.
- 8 Click OK to close the Runtime Configuration Properties dialog box.
- 9 Click OK to close the Project Properties dialog box.

The following sections describe running tests with each of the available test runners.

### See also

- [“Setting runtime configurations” on page 7-6](#)

## JBTestRunner

---

JBTestRunner provides a combination of text output and GUI indications of test status. JBTestRunner displays the current test hierarchy of test suites, test cases, and their test methods. You can navigate to a test method simply by clicking on it in the test tree. The cursor is positioned on the test method in the editor. JBTestRunner is JBuilder’s default test runner.

To run a test, right-click on a test case or test suite in the project pane and select Run Test from the context menu. Assuming you have not changed the default test runner in the Project Properties dialog box, the test will be run using JBTestRunner. If you have changed the default test runner and want to switch back to JBTestRunner, you can do so by following the steps for selecting a test runner that were described in [“Running tests” on page 13-13](#).

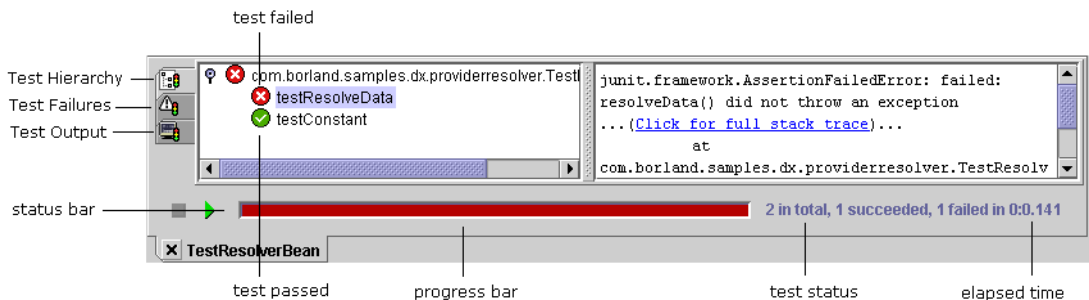
When you run tests, the results are displayed in the JBTestRunner page in the message pane. There are three views within this page: Test Failures, Test Hierarchy, and Test Output.

As tests run, JBTestRunner displays a progress bar which indicates the percentage of tests completed. This progress bar is green unless one or more tests have failed, in which case it is red. JBTestRunner also displays green check mark icons for successes and red X icons for failures or errors in the test hierarchy tree. In the case of a failure or error, JBTestRunner displays a stack on the right side of the Test Failures or Test Hierarchy page when the node for the failure or error is selected in the tree on the left

side. Click on a line in the stack and the point where an assertion failed will be highlighted in the editor.



As tests run, JUnitRunner's status bar indicates the number of tests run, the number of successes, failures, and errors. It also displays the time elapsed since starting the tests, including the time spent loading the test harness. The time elapsed is updated as each test completes.

If you right click any node in the Test Hierarchy tree or the Test Failures tree, a context menu appears which contains Run Selected and Debug Selected options. Use these options to run or debug a specific test when investigating a test failure.



## Test Hierarchy



The Test Hierarchy view is displayed by default, unless a test has failed. This view shows a tree listing test suites, test cases, and test methods. Each node in the tree has an icon next to it which indicates test status. A green check mark icon  indicates a passed test. A red X icon  indicates a failed test. This view is dynamically updated during a test run as the tests are run. Clicking a node in this tree causes the results for that node to be displayed in the right pane of the message view and also highlights the line of code causing a failure in the editor for a failed test or highlights the first line of a successful test method in the editor.

## Test Failures



The Test Failures view is accessed by clicking the middle tab on the left of the JUnitRunner page. The Test Failures view is displayed by default if a test has failed. This view displays a line for each test failure in the left pane. Clicking a line for a test failure displays more information about the failure in the right pane and also highlights the failure in the editor. If no tests have failed, this view will be empty.

## Test Output



The Test Output view is accessed by clicking the bottom tab on the left of the JUnitRunner page. This view displays any output generated by the tests, including exceptions.

## JUnit TextUI

---

JUnit's TextUI is a simple test runner that provides text-only output. JUnit has been integrated into JBuilder in such a way that when you run tests using JUnit's TextUI through the JBuilder IDE, you can simply click a line of output indicating a test failure in the message view and JBuilder's editor opens with the line of code that caused the failure highlighted.

## JUnit SwingUI

---

JUnit's SwingUI is a test runner that provides a GUI indicating test status and text messages indicating failures. Although you can run tests using SwingUI through the JBuilder IDE, you don't have the same ability to click a line of text indicating a failure and go right to that line in the editor that you have when using either JUnit's TextUI or JBuilder's JUnitRunner. The advantage of SwingUI is that you can review your test hierarchy and rerun one single test method at a time.

## Runtime configurations

---

A runtime configuration consists of VM parameters to be used and, in the case of a test, the test runner to use. To set the properties of a runtime configuration for running tests, go to Project | Project Properties, open the Run page, click the New, Copy, or Edit button, and click the Test tab. Here you can specify VM parameters to use when running your tests and select a test runner. In addition to the default runtime configuration, you can define additional runtime configurations by going to The Run page of the Project Properties dialog box or Run | Configurations. The Test Case and Test Suite wizards also let you set up a runtime configuration.

**Tip** You can also run tests using a runtime configuration for applications by invoking the `main()` method of the TextUI test runner. This might be useful if you wanted to write a script that invokes the test runner using command line arguments.

### See also

- [“Setting runtime configurations” on page 7-6](#)

## Defining a test stack trace filter

---

A test stack trace filter allows you to specify packages and classes to exclude from stack traces when running unit tests using JUnitRunner. Stack trace lines for excluded packages and classes are not displayed. This lets you concentrate on the stack trace information that's useful to you.

To specify a unit testing stack trace filter,

- 1 Select Project | Project Properties.
- 2 Select the Class Filters page of the Project Properties dialog box.
- 3 Select Unit Testing Stack Trace from the drop-down list.
- 4 Use the Add and Remove buttons to specify the packages and classes to exclude.
- 5 Click OK.

The unit testing stack trace filter excludes the following packages and classes by default.

- `junit.framework.*`
- `java.lang.reflect.Method`
- `com.borland.jbuilder.unittest.JBTestRunner`
- `sun.reflect.NativeMethodAccessorImpl`
- `sun.reflect.DelegatingMethodAccessorImpl`

Use the Add and Remove buttons on the Class Filters page of the Project Properties dialog box to edit this list.

## Debugging tests

---

Debugging unit tests is similar to debugging other code using JBuilder's debugger. The only difference is that when debugging a test the Test Hierarchy and Test Failures tabs from JBTestRunner are displayed in addition to the regular debugger UI. To debug a test, right-click any test in the project pane and select Debug Test from the context menu.

- Tip** When running tests using either JBTestRunner or TextUI, clicking an error in the test output highlights the line of code causing the failure in the editor. Clicking again in the left margin of the editor window next to the highlighted line of code sets a breakpoint on the line. You can then easily debug to the breakpoint.
- Tip** You can set a breakpoint on the exception thrown for a test failure. To do this, go to Run | Add Breakpoint | Add Exception Breakpoint and enter `junit.framework.AssertionFailedError` for the Class Name.

### See also

- [“JBTestRunner” on page 13-14](#)
- [Chapter 8, “Debugging Java programs”](#)





## Creating Javadoc from API source files

This is a feature of  
JBuilder SE and  
Enterprise

Javadoc is a tool created by Sun Microsystems to generate API documentation in HTML-formatted files. The generated HTML documentation is derived from class and method level comments that you enter into your API source files. The comments must be formatted according to Javadoc standards. For complete information about the Javadoc tool, go to the Javadoc Tool home page on Sun's website at <http://www.java.sun.com/j2se/javadoc/>.

JBuilder includes a number of features to support Javadoc generation. A wizard creates a documentation node that holds properties for a Javadoc run. This node is displayed in the project pane. Javadoc can be generated each time you build your project, using the current properties.

JBuilder also includes these other Javadoc-related features:

- A comment template that fills in parameters based on the class, interface, method, field, or constructor signature
- A template for adding `@todo` tags
- Reporting of Javadoc comment conflicts
- A Doc viewer to view the generated Javadoc
- "On-the-fly" Javadoc generation
- Documentation archiving with the Archive Builder

## Adding Javadoc comments to your API source files

---

You can add class and interface-level Javadoc comments as well as method, constructor and field-level comments. Javadoc comments are pulled out of your source files by the Javadoc tool and put into HTML-formatted documentation files.

A Javadoc comment starts with a begin-comment symbol (`/**`) and ends with an end-comment symbol (`*/`). Each comment consists of a description followed by one or more tags. If desired, you can use HTML formatting in your Javadoc comments. Follow these suggestions when entering comments:

- Indent the begin-comment symbol (`/**`) so that it lines up with the code being documented.
- Start subsequent lines of the comment with `*` (an asterisk). Indent these lines also.
- Start the descriptive text on the line after begin-comment symbol (`/**`).
- Insert a blank space before the descriptive text or the tag.
- Insert a blank comment line between the descriptive text and the list of tags.

An example of a Javadoc comment for a method is:

```
/**
 * Sets this check box's label to the string argument.
 *
 * @param label a string to set as the new label, or null for no label.
 */
```

In the generated HTML file, this comment displays as:

Sets this check box's label to the string argument.

**Parameters:**

label - a string to set as the new label, or null for no label.

Notice how Javadoc turned the `@param` tag into a heading. It also added the hyphen that separates the name of the parameter from its description. Additionally, it displayed the parameter name using a code font.

When you are writing the descriptive part of the comment, make the first sentence a summary. It should be a concise and complete description of the API item. The Javadoc tool copies the first sentence of the comment to the class, interface, or member summary table.

**Note** The Javadoc tool inherits comments for methods that implement or override other methods. In these cases, you do not need to duplicate method comments.

For more information on creating Javadoc comments, see “How to Write Doc Comments for the Javadoc Tool” at <http://www.java.sun.com/j2se/javadoc/writingdoccomments/index.html>.

### See also

- “Automatically generating Javadoc tags” on page 14-6
- “Javadoc Tags” (Windows platforms) at <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#javadoctags>
- “Javadoc Tags” (Solaris platforms) at <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#javadoctags>

## Where to place Javadoc comments

---

You can add Javadoc comments for classes, interfaces, methods, fields, and constructors. Place class and interface comments at the top of the file, after the `import` statements and immediately before the class or interface declaration statement. For example, the class comment for `com.borland.internetbeans.PageProducer.java` is:

```
package com.borland.internetbeans;

import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.servlet.ServletContext;
import javax.servlet.http.*;

/**
 * Generates markup text from a template file, replacing
 * identified spans with dynamic content from Ix
 * components.
 */
public class PageProducer implements Binder, Renderable, Cloneable,
Serializable {
    ...
}
```

You can enter class-level comments into the Class Javadoc Fields on the General page of the Project Properties dialog box. The comments you enter here will be added to every class or interface you create with a JBuilder wizard.

Method, field, and constructor comments are placed immediately before the method signature in the source class file. For example, if your source code contains the following methods:

```
public void addValues(Double valueOneDouble, Double valueTwoDouble) {
    double valueOneDoubleResult = valueOneDouble.doubleValue();
    double valueTwoDoubleResult = valueTwoDouble.doubleValue();
    addResult = (valueOneDoubleResult + valueTwoDoubleResult);
    addStringResult = Double.toString(addResult);
    addResultDisplay.setText(addStringResult);
}

public void subtractValues(Double valueOneDouble, Double valueTwoDouble) {
    double valueOneDoubleResult = valueOneDouble.doubleValue();
    double valueTwoDoubleResult = valueTwoDouble.doubleValue();
    subtractResult = (valueOneDoubleResult - valueTwoDoubleResult);
    subtractStringResult = Double.toString(subtractResult);
    subtractresultDisplay.setText(subtractStringResult);
}
```

you would add the Javadoc comments immediately before the method declaration. The resulting methods with Javadoc comments would look like this. Javadoc comments are in bold-face text.

```
/**
 * Adds Value One and Value Two and displays result.
 *
 * @param valueOneDouble The first value.
 * @param valueTwoDouble The second value.
 */
public void addValues(Double valueOneDouble, Double valueTwoDouble) {
    double valueOneDoubleResult = valueOneDouble.doubleValue();
    double valueTwoDoubleResult = valueTwoDouble.doubleValue();
    addResult = (valueOneDoubleResult + valueTwoDoubleResult);
    addStringResult = Double.toString(addResult);
    addResultDisplay.setText(addStringResult);
}

/**
 * Subtracts Value One and Value Two and displays result.
 *
 * @param valueOneDouble The minuend.
 * @param valueTwoDouble The subtrahend.
 */
public void subtractValues(Double valueOneDouble, Double valueTwoDouble) {
    double valueOneDoubleResult = valueOneDouble.doubleValue();
    double valueTwoDoubleResult = valueTwoDouble.doubleValue();
    subtractResult = (valueOneDoubleResult - valueTwoDoubleResult);
    subtractStringResult = Double.toString(subtractResult);
    subtractresultDisplay.setText(subtractStringResult);
}
```

## Javadoc tags

You can use the following tags in your Javadoc comments. Some of the tags, such as `@param` or `@return` are automatically included in a Javadoc run that is initiated through JBuilder. The wizard used to generate Javadoc allows you to select several other tags on the Specify doclet command-line options page. You can also enter other tags into the Additional Options field on the same page, forcing the wizard to include those comment tags in the generated files.

The following table lists and describes Javadoc tags. It indicates what Javadoc doclet the tags will be processed for (not all tags will be processed by the JDK 1.1 doclet, for example). It also explains what part of your class file the tag can be applied to. For more information on individual tags, see:

- Windows users - “Javadoc Tags” at <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#javadoctags>
- Solaris users - “Javadoc Tags” at <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#javadoctags>

**Table 14.1** Javadoc tags

Tag	Description	JDK 1.1 doclet	Std doclet	Kind of tag
<code>@author <i>name</i></code>	Adds an Author entry with the specified <i>name</i> to the generated docs when the <code>@author</code> option is selected on the Specify doclet command-line options page of the wizard used to generate Javadoc.	X	X	Overview, package, class, interface
<code>{@docRoot}</code>	The relative path to the generated document's (destination) root directory from any generated page. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Overview, package, class, interface, field
<code>@version <i>version-number</i></code>	Adds a Version subheading with the specified <i>version-number</i> to the generated docs when the <code>@version</code> option is used on the Specify doclet command-line options page of the wizard.	X	X	Overview, package, class, interface
<code>@param <i>parameter-name</i> <i>description</i></code>	Adds a parameter to the Parameters subheading. Automatically included in generated docs.	X	X	Constructor, method
<code>@return <i>description</i></code>	Adds a Returns subheading with the <i>description</i> text. Automatically included in generated docs.	X	X	Constructor, method
<code>@deprecated <i>deprecated-text</i></code>	Adds a comment indicating that this API has been deprecated and should no longer be used, even though it may still work. This option can be set on the Specify doclet command-line options page of the wizard.	X	X	Package, class, interface, field, constructor, method
<code>@exception <i>class-name</i> <i>description</i></code>	A synonym for <code>@throws</code> . Automatically included in generated docs.	X	X	Constructor, method

**Table 14.1** Javadoc tags (continued)

Tag	Description	JDK 1.1 doclet	Std doclet	Kind of tag
<code>@throws</code> <i>class-name description</i>	A synonym for <code>@exception</code> . Adds a Throws subheading to the generated docs with the <i>class-name</i> of the exception that can be thrown. Automatically included in generated docs.		X	Constructor, method
<code>@see</code> <i>reference</i>	Adds a See Also subheading to the generated docs. Automatically included in generated docs.	X	X	Overview, package, class, interface, field, constructor, method
<code>@since</code> <i>since-text</i>	Adds a Since heading with the specified <i>text</i> to the generated docs. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.	X	X	Overview, package, class, interface, field, constructor, method
<code>@serial</code> <i>field-description</i>	Describes a default serializable field. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Field
<code>@serialField</code> <i>field-name field-type field-description</i>	Documents an <code>ObjectStreamField</code> component of a serializable class <code>serialPersistentFields</code> member. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Field
<code>@serialData</code> <i>data-description</i>	Documents the type and order of data in the serialized form. The wizard does not automatically set this option; you need to enter it into the Additional Options field on the Specify doclet command-line options page.		X	Constructor, method
<code>{@link}</code> <i>package.class#member label</i>	Inserts an in-line link with the <i>label</i> as visible text. Automatically included in the generated docs.		X	Overview, package, class, interface, field, constructor, method

## Automatically generating Javadoc tags

JBuilder can automatically generate some of these tags for you. When the cursor is on a line before a class or interface declaration, typing the

begin-comment symbol (`/**`) and pressing *Enter* inserts the following template into your code:

```
/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2001</p>
 * <p>Company: </p>
 * @author
 * @version 1.0
 */
```

You need to fill in the fields as you wish.

**Note** This template (for classes and interfaces) can be filled in for a new project on the Specify General Project Settings page of the Project wizard as you're creating the project. You can also change these values at any time on the General page of the Project Properties dialog box.

When the cursor is before a method, field, or constructor signature, entering `/**` inserts the following template. Note that only those tags that are used in the signature are displayed in the expanded comment template.

```
/**
 *
 * @param
 * @throws
 * @returns
 */
```

JBuilder completes the tag for you by filling in the name of the parameter or exception. For example, for the following method signature:

```
public void addValues(Double valueOneDouble, Double valueTwoDouble)
```

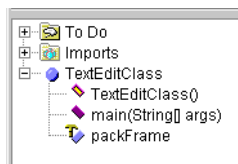
entering `/**` creates the following Javadoc comment:

```
/**
 *
 * @param valueOneDouble
 * @param valueTwoDouble
 */
```

## Javadoc @todo tags

Javadoc `@todo` tags are useful for adding reminders about what needs to be done to an area of code. These tags are placed inside of Javadoc comments. These `@todo` tags appear in JBuilder's structure pane in a `ToDo` folder.

**Figure 14.1** `ToDo` folder in structure pane



To add `@todo` tags to your code,

- 1 Type `todo` at the appropriate indentation level in the editor.
- 2 Press **Ctrl+J** to expand the template in your code:

```
/** @todo */
```

Some of JBuilder's wizards generate `@todo` tags as reminders to add code to the stub code that the wizard generates.

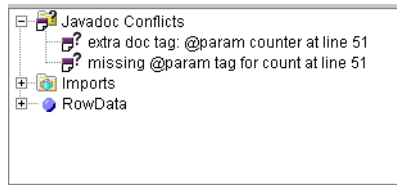
## Conflicts in Javadoc comments

---

A Javadoc conflict occurs when the tagging in a Javadoc comment does not match the method signature or if no argument is provided in tags such as `@param`. For example, if the method signature contains two parameters, and the comment only contains one, a conflict is reported.

In JBuilder, Javadoc conflicts are reported at the top of the structure pane in a folder called `Javadoc Conflicts`. Expand the folder and click the conflict to go to the method signature where the conflict occurred.

**Figure 14.2** Javadoc conflicts in structure pane



**Note** Javadoc conflicts are not reported until all syntax errors in the Errors folder are resolved.

## Generating the documentation node

---

JBuilder's Javadoc wizard generates a documentation node in the project pane. This node stores the properties for a Javadoc run. Properties include the format of the output, what packages are documented, and what output for those packages is generated. To change Javadoc properties after you create the node, right-click the node and choose **Properties**.

When you create the node, you can choose to create Javadoc files every time you build your project. You can also create Javadoc only on demand by right-clicking the node and choosing **Make**.

To display the Javadoc wizard, choose **File | New**. On the **Build** page of the object gallery, double-click the Javadoc icon. You can also choose **Wizards | Javadoc**.



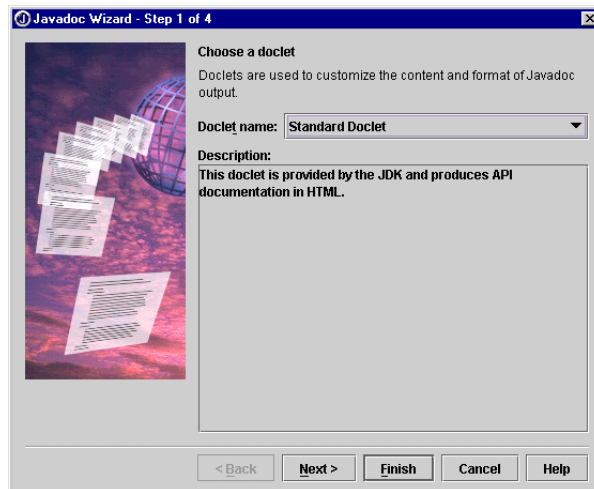
This wizard contains four steps. Options on the wizard include:

- The format of the Javadoc output.
- The name of the documentation node the wizard generates.
- The output directory.
- When to run Javadoc.
- The packages and scope to document.
- What output files are generated and what tags are processed.

## Choosing the format of the documentation

The first step of the wizard is where you choose the formatting of the Javadoc output. Output is controlled by a doclet, a Java class that specifies the contents and format of the HTML output files.

**Figure 14.3** Choose a doclet page



To choose the formatting of the output files,

- 1 To create output files in JDK 1.1 formatted Javadoc, choose the JDK 1.1 Doclet option from the Doclet Name drop-down list. This doclet creates output as HTML but does not include the additional level of detail provided with the Standard Doclet option. This option corresponds to the `-1.1` Javadoc tag. Note that this doclet is not available when running Javadoc using JDK 1.4.
- 2 To create output in JDK 1.4 formatted output, choose the Standard Doclet option from the drop-down list. This doclet produces API documentation in HTML format and includes more features than the JDK 1.1 output option, including:
  - Tables of fields and methods for a class or interface.
  - Package-level descriptions.

- Lists of inherited fields and methods.
- Lists of inner classes.
- A separate index per letter.
- Comments for the @use tag.
- Extensive navigation bar.

For an example of the Standard output, choose Help | Java Reference on the JBuilder main menu bar. Sun's JDK API reference documentation is displayed. It uses the JDK 1.4 doclet for formatting output.

The two doclets use different HTML naming conventions and directory structures. When displaying Javadoc, the Doc tab first looks for files formatted using the Standard Doclet. If this type of file is not found, JBuilder next looks for files formatted using the JDK 1.1 Doclet. For more information, see [“Viewing Javadoc” on page 14-20](#).

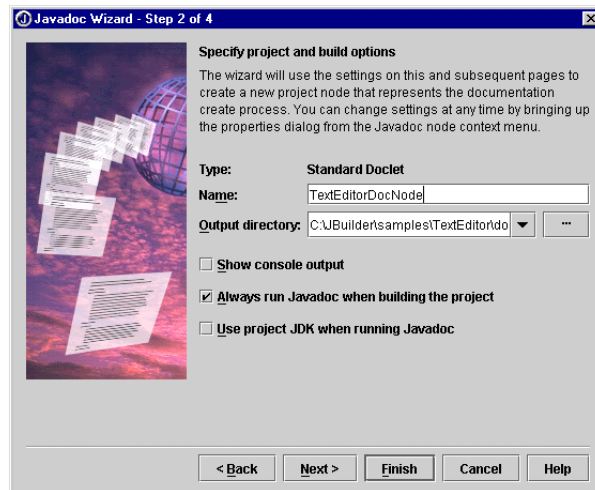
**Note** The Doclet Name option corresponds to the Javadoc **-doclet** option. The **-docletpath** option is explicitly set by the Javadoc wizard.

## Choosing documentation build options

The second step of the wizard is where you choose:

- The name of the documentation node that the wizard generates.
- The output directory.
- How Javadoc output is displayed.
- Javadoc build options.

**Figure 14.4** Specify project and build options page



To specify project and build options,

- 1 Enter the name for your documentation node in the Name field. This name is displayed in the project pane. By default, this is the type of doclet you selected in the previous step. You can change this to any descriptive name.
- 2 Enter the documentation output directory in the Output Directory field. This is the output path for the HTML files. The wizard uses the first directory path set on the Documentation tab of the Paths page of the Project Properties dialog box (Project | Project Properties). If you have not set a documentation path, the wizard suggests a `doc` directory in your project directory and creates it for you.

Use the ellipsis (...) button to browse to a new directory. If the directory does not exist, JBuilder creates it. You can also choose a previously selected path from the drop-down list. These paths are other documentation paths set in your project.

A single project can have multiple Javadoc paths, so that different packages, for example, can use different Javadoc options. Two projects should not share the same path. Each node has its own path.

This option corresponds to the `-d` Javadoc option. The `-sourcepath`, `-classpath`, and `-bootclasspath` Javadoc options are set by the wizard, based on project settings.

**Note** For maintenance purposes, Javadoc output should be kept in its own directory and not placed in the source or output directories. See [“Maintaining Javadoc” on page 14-22](#) for more information.

- 3 Choose the Show Console Output option to display Javadoc warnings in the message pane as the HTML files are being generated. This corresponds to the `-verbose` Javadoc option.

**Note** The Javadoc build stops if an error occurs. Errors are displayed in the message pane, regardless of the Show Console Output setting.

- 4 Choose the Always Run Javadoc When Building The Project option to generate Javadoc every time you build your project. You may want to turn this off when developing your project, as this can slow down compilation significantly.

If you don't choose this option, you can create Javadoc at any time by right-clicking the documentation node in the project pane and choosing Make.

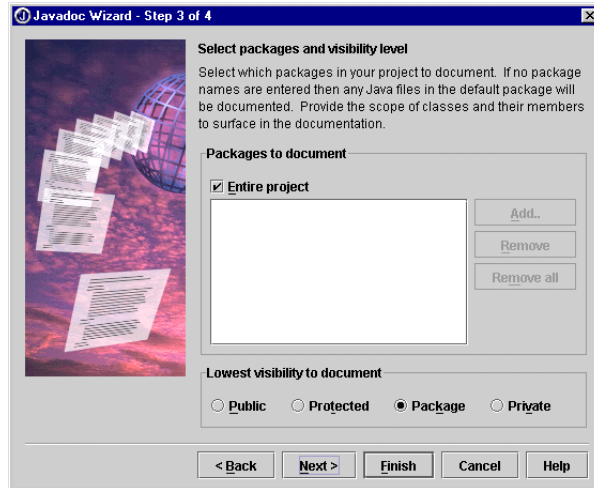
- 5 Choose the Use Project JDK When Running Javadoc to use the version of the JDK specified on the Paths page of the Project Properties dialog box. Otherwise, Javadoc is run using the JDK that hosts JBuilder.

**Note** The Javadoc wizard uses the value from the Encoding option on the General page of the Project Properties dialog box.

## Choosing the packages to document

This step of the wizard is where you choose what packages are documented and what scope of classes and members to surface in the documentation.

**Figure 14.5** Select packages and visibility level page



To select the package and visibility level,

- 1 Check the Entire Project option to document all packages in your project. This is on by default, so that all packages in your project are documented. Turn this option off to choose individual packages to document.
- Note** Source files in the default package are always documented.
- 2 Turn off the Entire Project option to document individual packages. The packages in your project are then displayed in the Packages To Document list.
    - Choose the Add button to add packages to this list. This displays the Select Packages To Document dialog box where you can choose individual packages in your project.
    - Select a package and choose the Remove button to remove a single package from the list.
    - Choose the Remove All button to remove all packages from the list.
  - 3 Choose one of the Lowest Visibility To Document options to select the scope of classes and members to include in the documentation:
    - Public - Includes only public classes and members in the documentation. This corresponds to the Javadoc **-public** option.

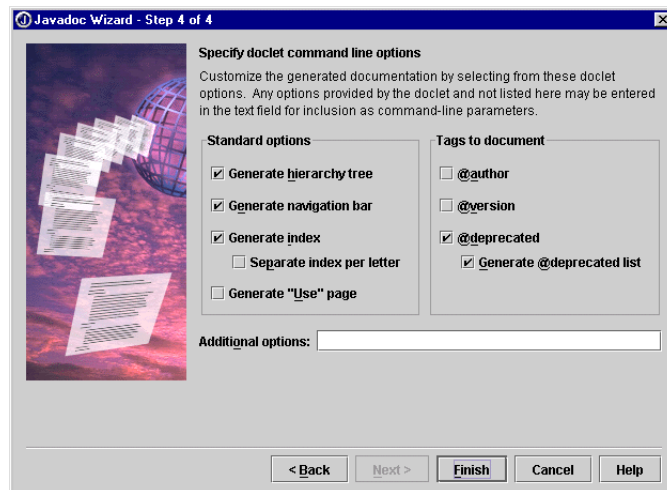
- Protected - Includes only protected and public classes and members in the documentation. This corresponds to the Javadoc **-protected** option.
- Package - Includes only package, protected and public classes and members in the documentation. This corresponds to the Javadoc **-package** option.
- Private - Includes all classes and members in the documentation, except those with private visibility. This corresponds to the Javadoc **-private** option.

**Note** If a source file has no elements with Javadoc comments that meet the lowest visibility requirement, the Javadoc tool doesn't generate an HTML file for that class.

## Specifying doclet command-line options

The last step of the wizard is where you define what output files are generated and what tags are processed. All selections on this page are optional; none are required for Javadoc to be generated.

**Figure 14.6** Specify doclet command-line options page



To specify command-line options,

- 1 Choose the Generate Hierarchy Tree option to generate the hierarchy tree for all classes in all packages. The hierarchy tree is a list of the hierarchies for all packages, classes, and interfaces in the documentation set. For the Standard doclet, a hierarchy tree is generated on a package-level basis. The hierarchy tree is stored in `overview-tree.html` in the root of your documentation path.

- 2 Choose the Generate Navigation Bar option to generate a navigation bar at the top of each HTML output file. This bar includes links to the next and previous package and class, the overview of all packages in the documentation set, the tree file, the index and the Javadoc-generated help topic.
- 3 Choose the Generate Index option to generate an index entry for each method and field, each package, each class, and each interface. The index is stored in `index-all.html` in the root of your documentation path. When this option is off, it corresponds to the **-noindex** Javadoc option.  
  
For JDK 1.4 Javadoc output, you can create links to index entries by letter. Choose the Separate Index Per Letter option. This corresponds to the **-splitindex** Javadoc option. The option is ignored for the JDK 1.1 doclet type, as the JDK 1.1 doclet always generates a separate index per letter.
- 4 Choose the Generate “Use” page option to generate one Use page for each package and a separate one for each class and interface. The package use file is called `package-use.html`; the class use file is `class-use/classname.html`. This page describes what packages, classes, methods, constructors, and fields use any part of the given class, interface, or package. The option is ignored for the JDK 1.1 doclet type.
- 5 Choose the `@author` option to generate documentation for `@author` tags in your source code. This option adds the author’s name to the generated Javadoc. One name or multiple names can be included in a single tag.
- 6 Choose the `@version` option to generate documentation for `@version` tags in your source code. This option adds the code version number to the generated Javadoc. This tag can apply to both a class or element in a class.
- 7 Choose the `@deprecated` option to generate documentation for `@deprecated` tags in your source code. This option adds a comment that the specified API element will be removed in a future version of the API.
- 8 Choose the Generate `@deprecated` List option to generate a list of `@deprecated` items. When this option is off, it corresponds to the **-nodeprecatedlist** Javadoc option.
- 9 Enter any additional options into the Additional Options field. These options are added to the command-line before the list of packages or files. Any options you set in this field override any options previously set in the wizard. Note that if you specify the **-locale** option, it always appears as the first command-line option.
- 10 Click Finish to create the documentation node.

The following table lists options that are not set by the wizard. These options can be set in the Additional Options field. The table indicates what Javadoc doclet the options will be processed for (not all options will be processed by the JDK 1.1 doclet, for example). For more information about Javadoc options, see:

- Windows users - “Javadoc options” at <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#javadocoptions>
- Solaris users - “Javadoc options” at <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#javadocoptions>

**Table 14.2** Options not set in the wizard

Option	Description	JDK 1.1 doclet	Std doclet
<b>-overview</b> <i>path\filename</i>	Specifies the file containing the text for the overview documentation.	X	X
<b>-help</b>	Displays the Javadoc help, which lists the Javadoc and command-line options.	X	X
<b>-Jflag</b>	Passes the <i>flag</i> directly to the runtime JDK that runs Javadoc. Do not include a space between the J and the <i>flag</i> . Use this option to increase memory for Javadoc, for example, <code>-J-Xms64m</code> . (The <code>-Xms</code> flag sets the size of initial memory. You can use it in conjunction with the <code>-Xmx</code> flag to increase available memory.)	X	X
<b>-locale</b> <i>language_country_variant</i>	Specifies the locale that Javadoc uses when generating documentation. Javadoc chooses the resource files of the specified locale for messages, such as strings in the navigation bar, headings for lists and tables, help file contents, and comments in the style sheet. It also specifies sorting for alphabetical lists.	X	X
<b>-doctitle</b> <i>title</i>	Specifies the title to be placed near the top of the overview summary file below the upper navigation bar.		X
<b>-windowtitle</b> <i>title</i>	Specifies the title to be placed in the <code>&lt;title&gt;</code> tag.		X
<b>-header</b> <i>header</i>	Specifies the header text to be placed at the top of each HTML-generated file to the right of the upper navigation bar.		X
<b>-footer</b> <i>footer</i>	Specifies the footer text to be placed at the bottom of each HTML-generated file, to the right of the lower navigation bar.		X
<b>-bottom</b> <i>text</i>	Specifies the text to be placed at the bottom of each HTML-generated file below the lower navigation bar.		X
<b>-link</b> <i>extdocURL</i>	Creates links to already existing Javadoc documentation for external referenced classes.		X
<b>-linkoffline</b> <i>extdocURL packagelistLoc</i>	Creates links to already existing Javadoc documentation for external referenced classes, where the package list file does not exist at the <i>extdocURL</i> location. (See <b>-link</b> .)		X
<b>-group</b> <i>groupheading</i> <i>packagepattern:package pattern:...</i>	Separates packages in the overview list into the specified groups.		X

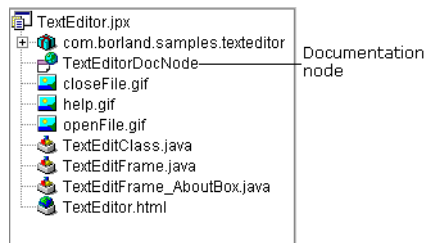
**Table 14.2** Options not set in the wizard (continued)

Option	Description	JDK 1.1 doclet	Std doclet
<b>-nosince</b>	Does not include comments in @since tags.		
<b>-nohelp</b>	Does not include the Help link in the upper and lower navigation bars.		X
<b>-helpfile</b> <i>path\filename</i>	Specifies the path of an alternate help file for the Help link in the navigation bar.		X
<b>-stylesheetfile</b> <i>path\filename</i>	Specifies the path of an alternate stylesheet file.		X
<b>-serialwarn</b>	Generates compiler warnings for missing @serial tags.		X
<b>-charset</b> <i>name</i>	Specifies the HTML character set for the generated documentation.		X
<b>-doencoding</b> <i>name</i>	Specified the encoding of the generated documentation.		X

**Important** If you enter options in the Additional Options field that have already been set in the wizard, your previous choices are overwritten.

## Generating the output files

Once you've set all options in the wizard and clicked the Finish button, the documentation node is created in the project pane.

**Figure 14.7** Documentation node in project pane

HTML files are not available until you generate them. To generate the HTML output files, you can,

- Build your project if you set the Always Run Javadoc When Building The Project option on the Specify project and build options page of the wizard.
- Right-click the documentation node and click Make. This option only builds Javadoc, it does not build your project. To delete HTML files in the configured directory, choose Clean. Choose Rebuild to do a Clean, and then a Build.



Output is displayed on the Compiler tab of the message pane if you set the Show Console Output option on the Specify project and build options page of the wizard.

Javadoc generates a set of HTML formatted files, based on the selected properties and placed in the selected output directory, usually the doc directory. Output files for the Standard Doclet include:

- A `classname.html` file for each class or interface in your project that contains documentation for the class or interface.
- A `package-summary.html` file for each package in your project that lists the classes and the package and provides overview information.
- An `overview-summary.html` file for the entire set of packages that is the documentation home page. This file is created only if your project contains two or more packages and you use the **-overview** option in the Additional Options field on the Specify doclet command-line options page of the wizard used to generate Javadoc.
- An `overview-tree.html` file for the class hierarchy for the entire set of packages.
- A `package-tree.html` file for the class hierarchy for each package.
- A `package-use.html` file for each package, class, and interface that lists what packages, classes, methods, constructors, and fields use any part of the given package, class, or interface. The `@use` option on the Specify doclet command-line options page of the wizard has to be set in order to generate this file.
- A `deprecated-list.html` file for all deprecated names. The `@deprecated` option on the Specify doclet command-line options page of the wizard has to be set in order to generate this file.
- A `serialized-form.html` file containing information about serializable and externalizable classes. The `@serial`, `@serialField` and `@serialData` tags need to be entered into the Additional Options field on the Specify doclet command-line options page of the wizard in order to generate this file.
- An `index-*.html` file that lists all class, interface, constructor, field, and method names, in alphabetical order. The Generate Index option on the Specify doclet command-line options page of the wizard has to be set in order to generate this file.

Javadoc also generates a number of support files, including:

- A `help-doc.html` file that describes how to use the navigation bar.
- An `index.html` file that creates the HTML frames.
- A number of `*-frame.html` files that list packages, classes, and interfaces used when HTML frames are displayed.
- A `package-list` file that is a text file, used by the `-link` options. It is not accessible through any links.
- A `stylesheet.css` file that controls the HTML display, based on the doclet you selected in the Choose a doclet page of the wizard.

For more information about the files that are generated, see “Generated Files” at:

- **Windows users:** <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#generatedfiles>
- **Solaris users:** <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#generatedfiles>

## Generating additional files

---

Javadoc automatically generates many output files for source `.java` files. You can also generate additional output files, such as package-level descriptive files and overview comment files, from auxiliary files.

### Package-level files

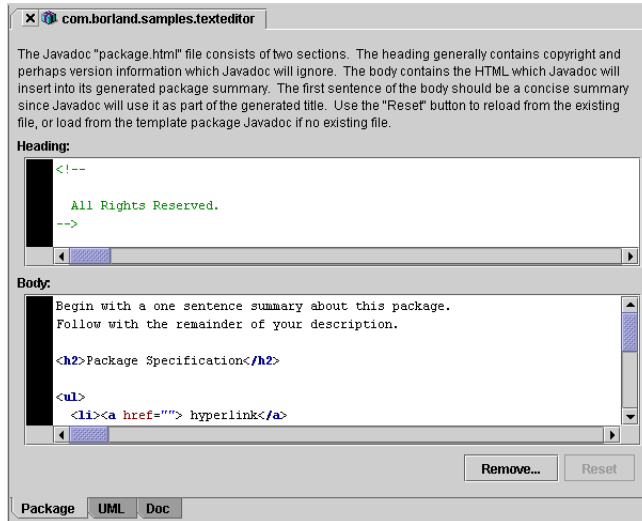
Each package can have its own documentation comment contained in its own source file. Javadoc merges this comment file into the package summary page it generates for each package in your project. This file must be called `package.html`, and it must be located in the package directory in the source tree along with the package’s class files. Javadoc automatically looks for this filename in this location.

The `package.html` file must contain a single documentation comment written in HTML. Do not include the Javadoc begin and end comment tags (`/**` and `*/`). Do not put a title or any other text between the `<body>` tag and the first sentence. The first sentence should be a summary.

JBuilder provides a package file editor that allows you to easily create, edit or remove the `package.html` file for individual packages in your project. The package file editor places the file in the correct location for Javadoc processing.

To use this editor,

- 1 Open your project. In the project pane, select the package you want to create a package file for.
- 2 Double-click the package. The package file editor is displayed on the Package tab of the content pane.



- 3 Enter any header text for the `package.html` file in the Heading section of the editor. This section generally contains copyright and version information. Javadoc will not process text in this section; it is left in a comment tag in the `package.html` file.
- 4 Enter body text in the Body section. The first sentence should be a concise one-sentence summary of the package. Follow this sentence with a complete description of the package. You can use the template to enter links to other packages and/or related documentation.

When you generate output files, the one-sentence description is displayed at the top of the package file. The remainder of the package description follows the Class Summary list. Note that you can create individual `package.html` files for each package in your project.

For more information about the package-level file, see “Package Comment Files” at:

- **Windows users:** <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#packagecomment>
- **Solaris users:** <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#packagecomment>

## Overview comment files

Each project can have its own overview comment file contained in its own source file. Javadoc merges this comment file into the overview page it generates for each all the packages in your project. You can name the file anything you like, usually `overview.html`. You can place it anywhere, usually at the top of your source tree.

The `overview.html` file must contain a single documentation comment, written in HTML. Do not include the Javadoc begin and end comment tags (`/**` and `*/`). Do not put a title or any other text between the `<body>` tag and the first sentence. The first sentence should be a summary.

For more information about the overview comment file, see “Overview Comment Files” at:

- **Windows users:** <http://java.sun.com/j2se/1.4/docs/tooldocs/win32/javadoc.html#overviewcomment>
- **Solaris users:** <http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html#overviewcomment>

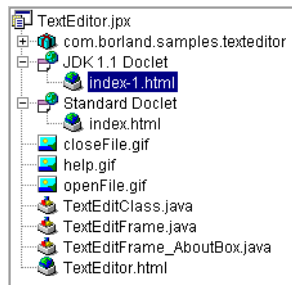
## Viewing Javadoc

---

There are several ways to view Javadoc once it has been built.

To view Javadoc for the entire project, expand the documentation node and double-click `index.html` (for output using the Standard Doclet) or `index-1.html` (for output using the JDK 1.1 Doclet).

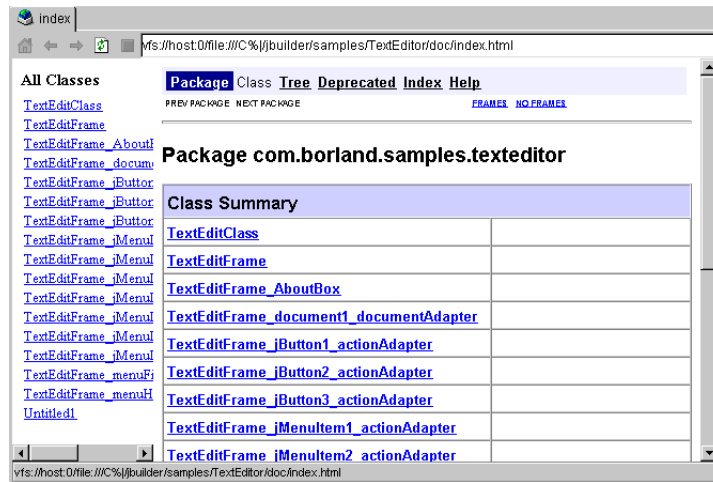
**Figure 14.8** Expanded documentation nodes



The index file opens in the View pane of the AppBrowser.

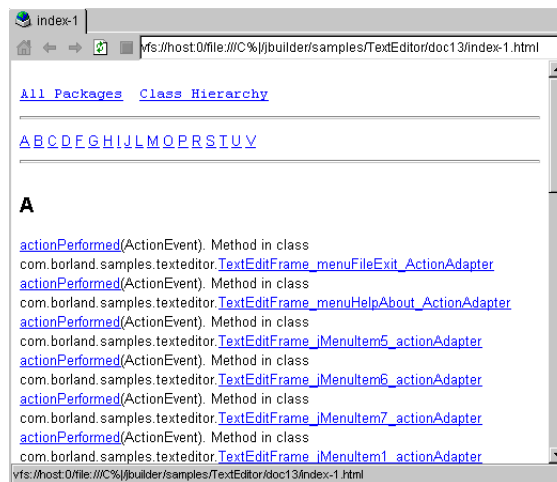
- For Standard Doclet output, packages and classes are listed in the left frame. The right frame displays summary tables.

**Figure 14.9** Index file output from Standard Doclet



- For JDK 1.1 Doclet output, an alphabetical index is displayed.

**Figure 14.10** Index file output from JDK 1.1 Doclet



You can also view Javadoc for an individual file by choosing the file in the project pane and selecting the Doc tab. To view Javadoc for an individual file from a UML class diagram, right-click the class name in the diagram and choose View Javadoc. The HTML file is displayed in the Help viewer.

For all views, you can use the upper or lower navigation bar, as well as links in the summary tables to navigate through the documentation.

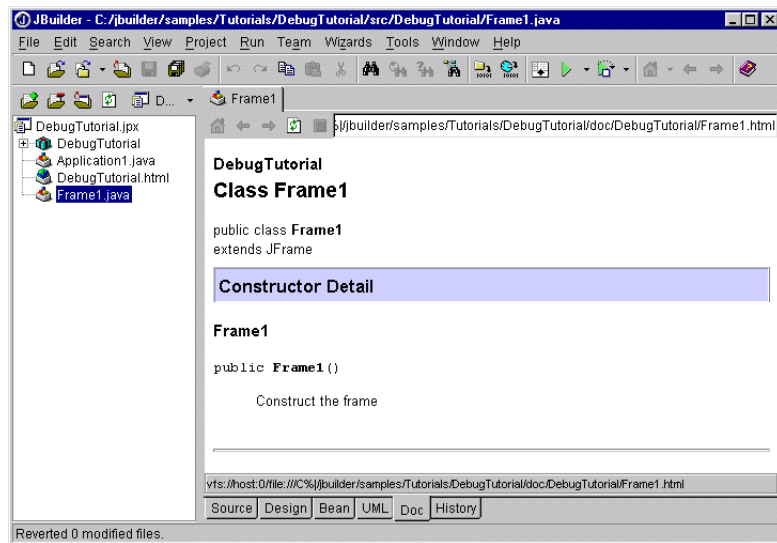
## How JBuilder displays Javadoc

When you choose the Doc tab to display Javadoc for a source file open in the editor, JBuilder first searches for HTML formatted standard or JDK 1.1 output, using the documentation path defined on the Doc tab of the Project Properties Paths page (Project | Project Properties).

- If only JDK 1.1 formatted output exists, that output is displayed.
- If both JDK 1.1 and standard formatted output exist, the Standard Doclet output is displayed.

If neither exists, JBuilder displays “on-the-fly” Javadoc that is generated directly from comments in the API source file. This allows up-to-date Javadoc to always be displayed for a source file, even if you have not yet created Javadoc. No links are available in this view.

**Figure 14.11** On-the-fly Javadoc output



## Maintaining Javadoc

The advantage of creating Javadoc in its own output directory, such as a doc directory, is that you can easily maintain it. To delete the HTML files in the documentation output directory, right-click the documentation node and choose Clean. The HTML files and the CSS files are removed. However, the directory structure is not deleted. Choosing Rebuild will clean the directory first and then rebuild the HTML files.

## Changing properties for the documentation node

---

Once the documentation node has been created, you can change node properties, including the name of the node, the output directory, and when Javadoc is generated. To change properties, right-click the documentation node in the project pane and choose Properties. In the Properties dialog box, choose the Node tab to change properties for the node. Choose the Javadoc tab to change what packages are documented. Choose the Doclet tab to change the doclet options.

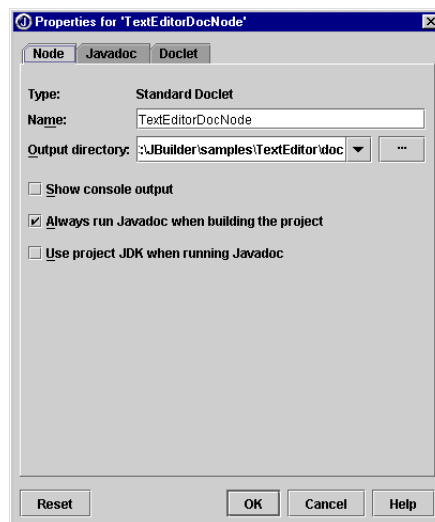
### Changing node properties

---

Use the Node tab of the Properties dialog box to change properties for the node. You can change the following options:

- Node name
- Output directory
- Display of console output
- When Javadoc is generated
- Which JDK to use (the project JDK or the one JBuilder is hosted on)

The Node page looks like this:



For more information, see [“Choosing documentation build options” on page 14-10](#).

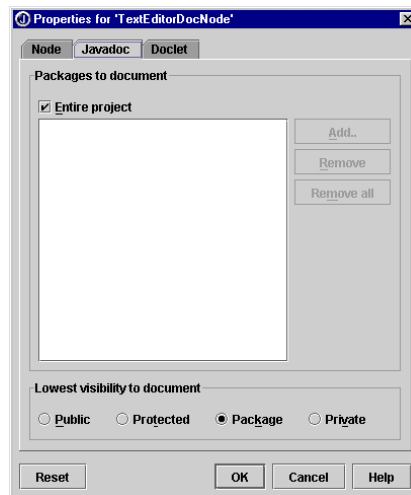
## Changing Javadoc properties

---

Use the Javadoc tab of the Properties dialog box to change what packages are documented. You can change the following options:

- Packages to document
- Lowest visibility to document

The Javadoc page looks like this:



For more information, see [“Choosing the packages to document” on page 14-12](#).

## Changing doclet properties

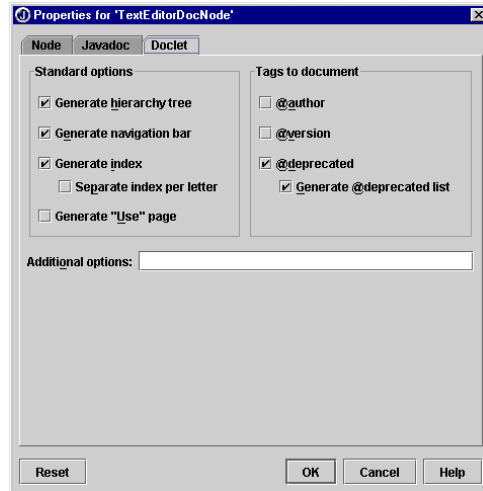
---

Use the Doclet tab of the Properties dialog box to change what options and tags are documented. You can choose:

- If the hierarchy tree file is generated
- If the navigation bar at the top of each generated file is displayed
- If an index is generated
- What tags are documented



The Doclet page looks like this:

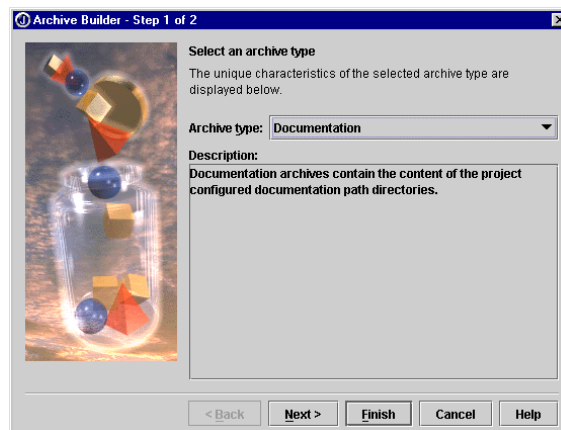


For more information, see [“Specifying doclet command-line options” on page 14-13](#).

## Creating a documentation archive file

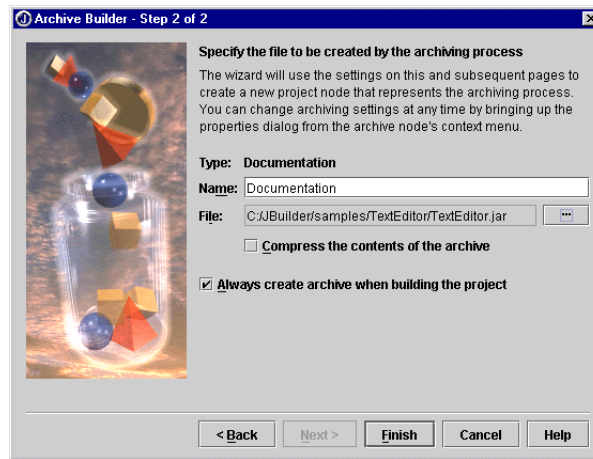
After your final Javadoc run, you can use the Archive Builder to create a documentation JAR file. This type of JAR file contains all files in the project’s documentation path directories, typically the `doc` directory. To create a documentation archive,

- 1 Choose Wizards | Archive Builder.
- 2 On the Select an archive type page of the Archive Builder, choose Documentation from the Archive type drop-down list. The Archive Builder looks like this:

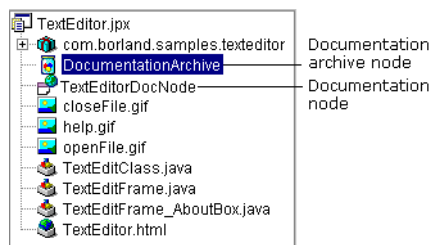


- 3 Click Next to go to the next step.
- 4 Enter the name for the Documentation node in the Name field. This node is displayed in the project pane when you click Finish.
- 5 Enter the name of the JAR file in the File field. This file should have a .jar extension. By default, it is placed in the root directory of your project.
- 6 Click Compress The Contents Of The Archive if you want the archive to be compressed. Use this option to make the JAR file as small as possible.
- 7 Click Always Create Archive When Building The Project to create the documentation JAR file each time you choose Make or Build.

This page of the Archive Builder should look like this:



- 8 Click Finish to close the wizard and create the DocumentationArchive node in the project pane. The project pane looks like this:



- 9 Choose Project | Make Project to make the project and create the JAR file.

The documentation JAR file is placed in the directory specified in the Archive Builder.

You can also create a source archive for the source files in your project using the Source archive type on the Select an archive type page of the Archive Builder.

For more information on using the Archive Builder, see [“Deploying with the Archive Builder” on page 15-17](#).

## Creating a custom doclet

---

You can create a custom doclet by extending the Javadoc wizard using the OpenTools API. For an example of a custom doclet, open the project `Doclet.jpj` in the `samples/OpenToolsAPI/wizards/doclet` folder of your JBuilder installation. A custom doclet does not need to produce HTML files. For example, custom tags can be used in conjunction with the Javadoc tool's ability to parse source files. The doclet could then generate XML files or additional Java files. The custom doclet must be placed in the `<jbuilder>/doclet` directory.

For more information about the OpenTools API, see “JBuilder OpenTools basics” in *Developing OpenTools*.



## Deploying Java programs

The Archive Builder and Native Executable Builder are features of JBuilder SE and Enterprise

Deploying a Java program consists of bundling together the various Java class files, image files, and other files needed by your program, and copying them to a location on a server or client computer where they can be executed. You can deliver the files separately, or you can deliver them in compressed or uncompressed archive files.

**Note** Throughout this document, any reference to a Java “program” implies a Java application, applet, Java Bean, or Enterprise Java Bean.

JBuilder SE and Enterprise editions provide the Archive Builder which assists you in deploying your program. In addition, the Native Executable Builder bundles an application JAR file with native executable wrappers for faster deployment. JAR files can also be created at the command line using Sun’s **jar** tool which is included in the JDK. JBuilder’s Archive Builder simplifies deployment by automatically gathering classes, resources, and libraries that your program needs and deploying the files to a compressed or uncompressed ZIP or JAR file. It also creates the JAR file’s `manifest`. The Archive Builder creates an archive node in your project, allowing easy access to the archive file and the `manifest`. At any time during development, you can make the archive file, rebuild it, or reset its properties. You can also view the contents of the archive, as well as the contents of the manifest file. See [“Deploying with the Archive Builder” on page 15-17](#) for more information.

The first step in deploying any program is to identify which project and library contents need to be included in the archive. This helps you determine what classes and resources, as well as dependencies, to include. Including all classes, resources, and dependencies in your archive creates a large archive file. However, you don’t need to provide your end-user with other files as the archive contains everything you need to run the program. If you exclude classes, resources, or dependencies, you’ll need to provide them to your end-user separately.

Deployment is a complex and advanced subject, requiring study to fully comprehend. JBuilder's Archive Builder reduces this complexity and helps you create an archive file that meets your deployment requirements.

### See also

- “Trail: Jar Files” in the Java Tutorial at <http://java.sun.com/docs/books/tutorial/jar/index.html>
- “Using JAR Files: The Basics” at <http://java.sun.com/docs/books/tutorial/jar/basics/index.html>
- Step 16 of the JBuilder tutorial, “Building a Java text editor” in *Designing Applications with JBuilder*

**Note** This document assumes you already understand the distinction between an applet (a program that runs within another context, typically a web browser) and an application (a stand-alone application that contains a `main()` method). For information on applets, browser issues, and JDK support, see the “Browser issues” topic in “Working with applets” in the *Web Application Developer's Guide*.

## Deploying to Java archive files (JAR)

---

Java programs can consist of many class files, plus various resource, property, and documentation files. A large program may consist of hundreds or even thousands of these files. Once your program is completed and ready to deploy, you need a convenient way to bundle all the classes and other files it uses into a single deployment set.

You can deploy the files individually or put them all into one easily deliverable archive file. You might even put a large program into a few archive files representing libraries and main programs. Compressed archive files give you the advantage of faster applet download time and less space required by your files on target server or system, and the disadvantage of slightly slower runtime speed.

The most efficient way to deliver, or deploy, a Java program is in a compressed JAR file. A JAR file also contains a manifest file and, potentially, signature files, as defined in the Manifest Specification. Some of the JAR's more advanced features, such as package sealing, package versioning, and electronic signing are made possible by the manifest file.

A JAR file (`.jar`) is basically a ZIP file with a different extension and with certain rules about internal directory structure. JavaSoft used the PKWARE ZIP file format as the basis for JAR file format.

**Note** JAR files are supported only in JDK 1.1 or later browsers. If you are deploying an applet to a JDK 1.0.2 browser, you need to use a ZIP archive file.

In addition to the class and resource files (placed in a package-appropriate directory structure), a JAR file must contain a manifest file and possibly class signature files.

Although you can technically place anything you want into an archive, the Java VM only looks for class files.

The HTML file from which an applet is loaded does not come from the archive. It is a separate file on the server. However, the JavaBeans specification indicates that HTML files documenting a bean can be placed into the archive.

## Understanding the manifest file

---

The `manifest` for a JAR file is a text-based file that includes information about some or all of the classes contained in that JAR file. In Java 2, it also contains information about which class in the JAR file is the runnable class.

The manifest file for any JAR file must be called `manifest.mf` and must be in the `meta-inf/` directory in the JAR file.

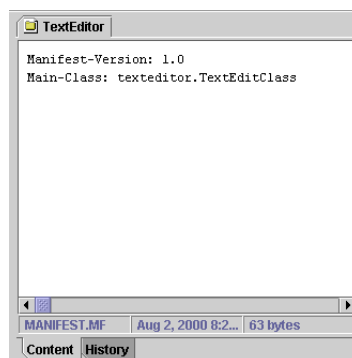
The default manifest file generated by the Archive Builder, available in JBuilder SE and Enterprise editions, puts the following two headers at the top of the file.

- `Manifest-Version: 1.0`

Tells you that the manifest's entries take the form of "header:value" pairs and that it conforms to version 1.0 of the manifest specification.

- `Main-Class: class-name`

This header is used for Application archive types. It indicates what class runs the application: the class containing the method `public static void main(String[] args)`.



The Main-Class header enables you to run your application from the JAR file using the **-jar** option to the Java Tools which launches the Java application from a command line.

There are other headers you can add to your manifest file which give you additional JAR file functionality, enabling the JAR file to be used for a variety of purposes.

### See also

- “Understanding the manifest” at <http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html>
- “Special purpose manifest headers” at <http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html#special-purpose>
- “JAR Manifest” at <http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR Manifest>
- “JAR Manifest - Main Attributes” at <http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#Main Attributes>
- “Command-line tools” at <http://java.sun.com/products/js2e/1.4/docs/tooldocs/tools.html#basic>
- “JAR Guide” at <http://java.sun.com/j2se/1.4/docs/guide/jar/jarGuide.html>

## Deployment strategies

---

There are two basic deployment strategies for delivering your application:

- Distribute the redistributable libraries with your JAR file and include them on the `CLASSPATH` at runtime, rather than putting the required classes from those libraries inside the JAR file. This is the easiest way to deploy and creates the smallest program JAR file. This is a reasonable choice, and one you might make if you are delivering multiple applications or applets to the same location and want them to share the libraries.

See the files, `<jbuilder>/license.html` and `<jbuilder>/redist/deploy.html`, for information about what you may or may not redistribute under the JBuilder product license.

- Create a JAR file using Sun’s **jar** tool, available in the JDK, or JBuilder’s Archive Builder, available in SE and Enterprise editions. JBuilder’s Archive Builder provides many options for gathering the classes, resources, and libraries your program needs. The options you choose depend on your deployment requirements, including space considerations, whether your program is an applet or a stand-alone



application, and how your users install your program. See [“Deploying with the Archive Builder” on page 15-17](#).

The deployment process in JBuilder can be summarized into the following basic steps:

- 1 Create and compile your code in JBuilder.
- 2 Create a JAR file using Sun’s **jar** tool or JBuilder’s Archive Builder.
- 3 Create an install procedure.
- 4 Deliver your JAR file, all necessary redistributable JAR files, and the installation files.

#### See also

- Sun’s **jar** tool at <http://java.sun.com/products/js2e/1.4/docs/tooldocs/tools.html#basic>
- [“Deployment quicksteps” on page 15-10](#)
- [“Deploying with the Archive Builder” on page 15-17](#)

## Using the JDK Java Archive Tool

---

Sun provides a way to create and modify a JAR file from the command line using the Java Archive Tool (**jar** tool) provided as part of the Java Development Kit. The **jar** tool is invoked with the **jar** command using the following basic format:

```
jar cf jar-file input-file(s)
```

For more information on creating and modifying JAR files, see [“Updating the contents of a JAR file” on page 15-7](#).

#### See also

- [“Using JAR Files: The Basics”](#) at <http://java.sun.com/docs/books/tutorial/jar/basics/index.html>

## Running a program from a JAR file

---

You can run a program archived in a JAR file from the command line. Add the JAR file to the **CLASSPATH**, for example, **CLASSPATH=user/username/jbproject/myapp/myjar.jar**, or add it to the **-cp** or **-classpath** command line option to **java.exe**. Give the full package name to the class.

```
java -classpath user/username/jbproject/myapp/myjar.jar mypackage.myclassname
```

In version 1.2 and above of the JDK software, you can add **-jar** to the **java** command to tell the interpreter that the application is packaged in the JAR

file format. The Java VM gets the information from the Main-Class: header in the manifest about which class to run.

```
java -jar jar-file
```

For example,

```
java -jar user/username/jbproject/myapp/myjar.jar
```

Of course, if you run the application from the same directory as the JAR file, you only have to type the following:

```
java -jar myjar.jar
```

### See also

- “Running JAR-packaged software” at <http://java.sun.com/docs/books/tutorial/jar/basics/run.html>
- “Modifying a manifest file” at <http://java.sun.com/docs/books/tutorial/jar/basics/mod.html>
- “Updating a JAR file” at <http://java.sun.com/docs/books/tutorial/jar/basics/update.html>

## Viewing archive file contents

---

To view a listing of the JAR file contents with the **jar** tool, use the command:

```
jar -tf jar-file
```

This also works for ZIP files:

```
jar -tf zip-file
```

**Note** You can also view archive contents in JBuilder. Add a JAR file to your project, double-click the JAR file in the project pane, expand the nodes in the structure pane, and double-click a file to open it in the editor. JAR files are read-only in the editor.

The JAR utility has many other uses. To see these, type **jar** for command-line help.

You can use most PKWARE-compatible ZIP file tools to examine or even modify JAR files. If the tool actually requires a ZIP extension, you can temporarily rename the JAR to ZIP.

**Note** Versions of WinZip prior to 6.3 contain a bug preventing them from extracting or viewing files from a valid JAR file. Later versions solve this.

## Updating the contents of a JAR file

---

Java 2 has a **u** option to `jar.exe` which you can use to update the contents of an existing JAR file by adding files. To use this command, type:

```
jar uf jar-file input-file(s)
```

where,

**u** = update an existing JAR file  
**f** = JAR file to update is on the command line  
**jar-file** = name of the existing JAR file to be updated  
**input-file(s)** = space-delimited list of files to add

Any files already in the archive having the same path name as a file being added are overwritten.

You can also use the **u** option with the **m** option to update an existing JAR file's manifest:

```
jar umf manifest jar-file
```

where,

**m** = update the JAR file's manifest  
**manifest** = name of manifest whose contents you want to merge into the manifest of the existing JAR file

### See also

- “Modifying a manifest file” at <http://java.sun.com/docs/books/tutorial/jar/basics/mod.html>
- “Updating a JAR file” at <http://java.sun.com/docs/books/tutorial/jar/basics/update.html>
- “How classes are found” at <http://java.sun.com/j2se/1.4/docs/tooldocs/findingclasses.html>
- The “JAR Files Trail” in the Java Tutorial at <http://java.sun.com/docs/books/tutorial/jar/>

## Deployment issues

---

The following questions need to be answered to determine the best deployment strategy:

- Is everything you need on the class path?
- Does your program rely on JDK 1.1.x or Java 2 (JDK 1.2 and above) features?

- Does the user already have the non-JDK Java libraries you use installed locally?
- Is this an applet or an application?
- Are there download time and/or server disk space limitations?

## Is everything you need on the class path?

---

Deployment problems are primarily about not having everything you need in the JAR file or on the class path. If you haven't added the required classes from a particular library to the JAR file, then make sure that the redistributable libraries are specified in the **-classpath** option of the Java command line or in your `CLASSPATH` environment variable. The **-classpath** option is the recommended way since you can set the class path individually for each application without affecting other applications and other applications can't modify its value.

**Tip** JBuilder tells you what your class path is when you run from the IDE. Mirror that at the command line and your program should work.

How you set the `CLASSPATH` environment variable depends on which operating system you are using.

### See also

- “Setting the `CLASSPATH` environment variable for command-line tools” on page B-2
- “Setting the class path” at <http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html>

## Does your program rely on JDK 1.1.x or Java 2 (JDK 1.2 and above) features?

---

If you are developing an applet, this might be an issue as some users may be using web browsers which have not been updated to support applets written with JDK 1.1.x or later features, such as Swing. See “Working with applets” in the *Web Application Developer's Guide* for more information.

JDK 1.0.2-compliant browsers do not support JAR archives, therefore if you have written a JDK 1.0.2-compliant applet and want to deploy it, be sure to create a ZIP archive format.

## Does the user already have Java libraries installed locally?

---

If your program uses components that rely on non-JDK libraries, you need to supply them to the user in the JAR file. You'll find the JBuilder redistributable files in the `<jbuilder>/redist/` directory. All JDK redistributable files are in the `<jbuilder>/<jdk>/lib/` and `<jbuilder>/<jdk>/jre/lib/` directories. `<jdk>` represents the name of the JDK directory.

**Note** When deploying with JBuilder's Archive Builder, available in SE and Enterprise editions, you can include these libraries by selecting the Include Required Classes And All Resources option on Step 4 of the wizard. This option ensures that the required classes, as well as all resources (including those from third-party libraries), are included in your program's JAR file. The Archive Builder never includes the JDK in your archive. It assumes that the JDK classes already exist on the target computer in the form of an installed JDK, Java runtime environment, or Java Plug-in or that you are providing it in your installation. See ["Deploying with the Archive Builder" on page 15-17](#).

If you are certain that your users have these archives in their environment, either because the users already have them installed, or because you have provided it to them using some kind of installation process, then you can deliver applications and applets that do not have to contain these packages.

If you are not certain your users have these libraries, you need to provide them. This is particularly true in the case of applet deployment. When you deploy your applet, you need to deploy these libraries and any other needed resources to the server.

**Important** In JDK 1.1.x, the Swing/JFC classes were **not** delivered as part of the JDK. If you are writing programs that use any of these JDKs, you must download and deliver `swingall.jar` which contains those files.

### See also

- ["Redistribution of classes supplied with JBuilder" on page 15-15](#)

## Is this an applet or an application?

---

Your deployment strategy for applications is different from applets. From strictly a deployment standpoint, the main difference between the two is as follows:

- For an application, your user needs to use `java.exe` (from the Java Runtime Environment) to run the classes or JAR files you have provided, either directly from the server or after installing them locally. If the user does not have the necessary JRE files, you must include them in your deployment set.

- For an applet, you assume the user is using an Internet browser or an applet viewer and that you have an HTML page containing an `<applet>` tag that references the classes you want to run. In this case, you must make sure the user's browser supports the version of the JDK that you used to develop your applets, and if not, provide them with the correct Java Plug-in from Sun to use with their browser. For more information, see "Working with applets" in the *Web Application Developer's Guide*.

**Important**

Before you plan to use the Java Plug-in from Sun, be sure to read all their documentation on the issues involved. There are several versions of the Java Plug-in and the HTML Converter. Because the plug-in versions are incompatible with each other, you can only have one plug-in installed at a time. Use of the Java Plug-in is recommended only in a controlled environment, such as an intranet situation, where you know which browser version is being used and which JDK it supports.

For more information, see the Java Plug-in, found at <http://java.sun.com/products/plugin/>.

## Download time

---

One of the first questions you have to answer is whether to place your program into an archive. The biggest factors in this decision are download time and program size, especially for applets.

One of the advantages of using the Archive Builder, available in JBuilder SE and Enterprise editions, to create an archive is that only the necessary files are included. The Archive Builder identifies all of the classes the project needs to use, and bundles them into one archive. This allows for efficient download time, disk usage, and network socket consumption when the application or applet is launched from a browser.

Until you become more familiar with Java classes and their dependencies, your JAR files may need to be larger to make sure everything you need is included. As your knowledge increases, you'll be able to trim down the size of your JAR files by only including specific classes and dependencies in your project and in your JAR file.

## Deployment quicksteps

---

Deployment steps vary according to what you are deploying. In brief, the following quicksteps describe how to deploy applications, applets, and JavaBeans. For more information on the Archive Builder, available in JBuilder SE and Enterprise, see "[Deploying with the Archive Builder](#)" on [page 15-17](#).

## Applications

---



- 1 Add all the resource files and dynamically loaded classes your application needs to your project using the Add Files/Packages button on the project pane toolbar.

Optional: Use the Resource Strings wizard, available in JBuilder SE and Enterprise, to move your strings to a resource bundle. This makes it easier to localize your application for a different locale.

- 2 Compile the application.
- 3 Use JBuilder's Javadoc wizard, available in JBuilder SE and Enterprise, to create appropriate documentation or use another means to create appropriate documentation for your developer customers.
- 4 Create a JAR file using Sun's **jar** tool, JBuilder's Archive Builder, or the Native Executable Builder.
- 5 Copy the JAR file to the target server or installation directories.
- 6 Create an installation procedure that makes the necessary folders and subfolders on the end user's computer and places the files in those folders. This procedure should also modify the `CLASSPATH` environment variable as needed in the end user's environment to specify the correct `CLASSPATH` for finding the Java classes.

**Note** If you're creating a Native Executable or Executable JAR archive type with the Archive Builder or Native Executable Builder, you can customize the installation procedure by modifying the configuration file that launches the executable.

**Note** If users need to have certain Java classes or archives already installed on their machine, you may need to provide a convenient means for them to download those files and add them to their local `CLASSPATH`.

- 7 Tell your users how to start your application (for example, by double-clicking on an icon you've provided, or running a shell script from the terminal window).

### See also

- Step 16 of the JBuilder tutorial, "Building a Java text editor" in *Designing Applications with JBuilder*

## Applets

---

Because browser JDK support varies, creating and deploying applets becomes quite complicated. The steps below are intended only as a broad guideline. To learn about all the applet and browser issues, see "Working with applets" in the *Web Application Developer's Guide*.



- 1 Add all the resource files and dynamically loaded classes your applet needs to your project using the Add Files/Packages button on the project pane main toolbar.  
  
Optional: Use the Resource Strings wizard, available in JBuilder SE and Enterprise, to move your strings to a resource bundle. This makes it easier to localize your applet for a different locale.
- 2 Compile the applet.
- 3 Create a JAR or ZIP file using Sun's **jar** tool, a ZIP utility, or JBuilder's Archive Builder.
- 4 Add the `archive` attribute with the name of the JAR file inside the `<applet>` tags in the applet HTML file. If you have multiple JAR files, list them separated by commas: `archive="file1.jar, file2.jar, file3.jar"`. Some earlier browsers do not support multiple listings and only accept ZIP files.
- 5 Check the `codebase` and `code` values for accuracy.  
  
The `codebase` attribute states where classes are located in relation to the location of the HTML file. If the `codebase` attribute is set to a value of `" "`, `"/"`, or `"/"`, the classes must be located in the same directory as the HTML file. If the classes are members of a package, the classes must be in a subdirectory of the HTML file's directory that matches the package structure. When the classes are in the same directory as the HTML file, the `codebase` attribute is optional, since this is the default. If the classes are located in a different directory than the HTML file, the `codebase` attribute is required and must specify the class files' directory relative to the HTML file's directory.  
  
The `code` value must be the fully qualified class name of the applet: package name + class name.
- 6 Launch your web browser and open your local HTML file for initial testing.
- 7 Copy the archive or the deployment file set to the target server.
- 8 Check that all case in class, package, archive, and directory names in the `<applet>` tag exactly match the names on the server.
- 9 Clear the classpath to eliminate any libraries and JDKs on your path. You need to assume your users have none of these on their class paths.



- 10 Test by way of the web server by pointing the browser at the URL that represents the HTML file on the target server.

**Note** Make sure you copy the correct HTML file to the target folder and put the JAR file in the same folder. A JBuilder project might contain more than one HTML file. If there is more than one HTML file, choose the HTML file containing the `<applet>` tag.

Optional: If users need to have certain Java classes or archives already installed on their machine, you may need to provide a convenient means for them to download those files and add them to their local CLASSPATH.

**Important** JDK 1.0.2-compliant browsers do not support JAR archives. Be sure to create a ZIP file instead.

### See also

- Step 7 of the JBuilder tutorial, “Building an applet” in *Introducing JBuilder*

## JavaBeans

---



- 1 Add all the resource files and dynamically loaded classes your application needs to your project using the Add Files/Packages button on the project pane main toolbar.

Optional: Use the Resource Strings wizard, available with JBuilder SE and Enterprise, to move your strings to a resource bundle. This makes it easier to localize your bean for a different locale.

- 2 Compile the bean(s).
- 3 Use JBuilder’s Javadoc wizard, available in JBuilder SE and Enterprise, to create appropriate documentation for the beans or use another means to create appropriate documentation for your developer customers.
- 4 Create a JAR file using Sun’s **jar** tool, a ZIP utility, or JBuilder’s Archive Builder. You can include the documentation in the archive if this is a development-time version, or omit it if it is a redeployable version. To include the documentation in the archive using the Archive Builder, make sure it is a project node by adding it to your project before you deploy.
- 5 Provide the JAR file(s) to your customers.

**Note** If your bean requires any of the JBuilder libraries, see [“Redistribution of classes supplied with JBuilder” on page 15-15](#).

## Deployment tips

---

Basic tips for making the deployment process successful include:

- Keep images in a subdirectory (typically called `images`) relative to the classes that use them. Do the same for any other resource files.
- Use only relative paths (for example, `images/logo.gif`) to refer to image files and other files used by your application or applet.
- Use packages, no matter how big or small an applet or application is.

## Setting up your working environment

---

The key to combining development and deployment is managing what files go where. As a rule, it's a good idea to keep your source code deliverables in a different hierarchy than your development tools. This keeps the irreplaceable items out of harm's way.

You'll find the deployment process easier if you make your project working environment reflect the reality of a deployed applet or application, bringing your development a little closer to instant deployment. Start your project with a JBuilder wizard, because it puts everything where it should be. Once you have your project built and running in this situation, you can create a JAR file using Sun's `jar` tool or JBuilder's Archive Builder, available in JBuilder SE and Enterprise editions. After creating the archive, you can test your program at the command line. Be sure to test your applets with any and all the browsers you need to use.

**Tip** If you are creating an applet, you can use a copy of the actual HTML page that is on your web site in your local project, instead of a simpler test one. That makes the external testing a little more realistic. When you're satisfied you can upload the entire directory to your Web site.

## Internet deployment

---

If you are deploying your program to a remote site by FTP, such as an Internet service provider, the basic procedure for deployment remains the same. However, you need to use an FTP utility to transfer the files, following directions provided by your web site provider.

**Important** Be sure to transfer archives and class files as binary files. An improper transfer to the Internet site causes `java.lang.ClassFormatError` exceptions.

Quite often the directory structure of your site as seen through FTP isn't quite the same as the URL with which your users access it. Your provider can tell you where your web site's root directory is and how to transfer files there via FTP. Most shareware and commercial FTP programs let you

create directories as well as copy files, so all the steps above should apply, although with a different file transfer mechanism.

For information on deploying web applications, see “Deploying your web application” in the *Web Application Developer’s Guide*.

## Deploying distributed applications

---

When deploying distributed applications, JBuilder’s Archive Builder collects your stubs and skeletons into a JAR file. You must install your ORB on each machine that runs a client, middle tier, or server CORBA program. If you are using the VisiBroker ORB, see the deployment topic in the Borland Enterprise Server documentation or see your application server documentation.

## Redistribution of classes supplied with JBuilder

---

The JBuilder redistributable JAR files are located in the `<jbuilder>/redist/` directory. The redistributable archive files for the JDK are in the `<jbuilder>/<jdk>/lib/` directory.

If you are creating your archive with JBuilder’s Archive Builder, available in JBuilder SE and Enterprise editions, it detects all the resource files, classes, and libraries you need. The Archive Builder does not include the JDK in your archive. It assumes that the JDK classes already exist on the target computer in the form of an installed JDK, Java runtime environment, or Java Plug-in, or that you are providing it in your installation.

**Important** In the case of Java 1.1.1, however, the Swing/JFC classes are not part of the core JDK and won’t be detected by the Archive Builder. If you are writing a program that uses these JDKs, be sure to download and deliver `swingall.jar`.

If you deploy an applet, you do not need to deliver the JDK JRE classes, because they are supplied by the browser at runtime. However, you do need to make sure the version of the JDK used by the applet and the version used by the browser match. In an intranet situation, you can use Sun’s Java Plug-in to provide the current JDK.

For information on browser issues and JDK support, see “Browser issues” in “Working with applets” in the *Web Application Developer’s Guide*.

**Note** Typically, the only thing on the `CLASSPATH` when running an applet is the Java classes. If a third-party library is on the `CLASSPATH` as classes or an archive, and some of those classes are also deployed in the applet or JAR file, the `CLASSPATH` copy would be preferred since the System class loader

can load it. Whichever one is listed first satisfies the VM and it is as if the “second” one is not listed at all.

Please examine the files `<jbuilder>/license.txt` and `<jbuilder>/redist/deploy.txt` for information about what you may or may not redistribute under the JBuilder product license. `<jbuilder>` represents the name of the JBuilder directory.

### See also

- **Java Runtime Environment - “Notes for Developers”** at <http://java.sun.com/j2se/1.4/runtime.html>
- **“The JAR Trail” in the Java Tutorial** at <http://java.sun.com/docs/books/tutorial/jar/>
- **“JAR Guide”** at <http://java.sun.com/j2se/1.4/docs/guide/jar/jarGuide.html>

## Additional deployment information

---

You can find additional information at the following URLs:

- **Java Runtime Environment - “Notes for Developers”** at <http://java.sun.com/j2se/1.4/runtime.html>
- **“Trail: Writing Applets” in the Java Tutorial**, which discusses basic applet considerations such as security, at <http://java.sun.com/docs/books/tutorial/applet/index.html>
- **“Trail: Security in Java 2 SDK 1.2” in the Java Tutorial**, which Discusses general security APIs and issues, at <http://java.sun.com/docs/books/tutorial/security1.2/index.html>
- **“The JAR Trail” in the Java Tutorial** at <http://java.sun.com/docs/books/tutorial/jar/>
- **“Understanding the manifest”** at <http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html>
- **“JAR Guide”** at <http://java.sun.com/j2se/1.4/docs/guide/jar/jarGuide.html>
- **Sun developer training and tutorials** at <http://developer.java.sun.com/developer/onlineTraining/index.html>
- **“Writing advanced applications,”** which summarizes problems and solutions related to having different versions of the Java platform installed on your system, at <http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/version.html>

- “The Extension Mechanism Trail” of the Java Tutorial at <http://java.sun.com/docs/books/tutorial/ext/index.html>. Explains the new Java 1.2 extension classes which make custom APIs available to all applications running on the Java platform. The extension mechanism enables the runtime environment to find and load extension classes without the extension classes having to be named on the class path.

## Deploying with the Archive Builder

---

This is a feature of  
JBuilder SE and  
Enterprise

The Archive Builder speeds up the process of creating your deployment set. It searches your project for classes and resources and gives you the opportunity to customize the JAR file contents before it archives the files into a JAR file with the appropriate manifest file.

### The Archive Builder and resources

---

The Archive Builder automatically recognizes certain file types as resources as specified on the Resource tab of the Build page in Project Properties. JBuilder copies resources, such as images, sound, and properties files, from the source path to the output path when compiling. The output path, which is set on the Paths page of the Project Properties dialog box, contains the `.class` files created by JBuilder when it compiles a program. See “[How JBuilder constructs paths](#)” on page 4-9 for more information on paths.

**Note** JBuilder can’t determine which resources are required based on the Java code. The resources must be in the project.

To see a list of the default settings by file extension, see the Resource tab of the Build page. You can change JBuilder’s default settings and specify individual files or file extension types to be copied to the output path during compile. See “[Selective resource copying](#)” on page 6-25.

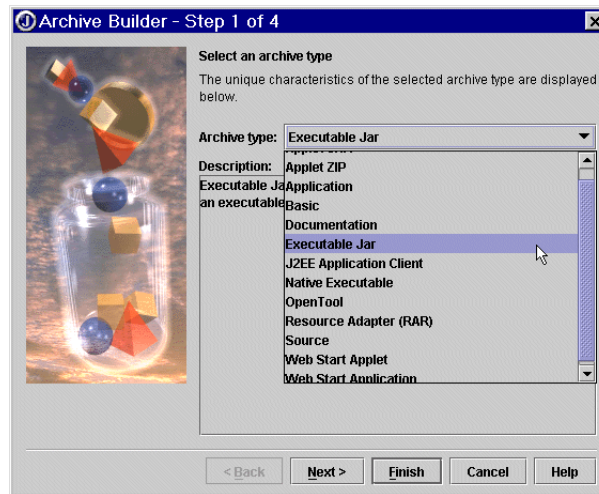
If you have file types in your project that JBuilder doesn’t recognize, you can add them as generic resource files, then specify them to be copied to the output path. For more information, see “[Adding unrecognized file types as generic resource files](#)” on page 6-27.

### Selecting an archive type

---

The first step of the Archive Builder lets you select what type of archive you want to create: Applet JAR, Applet ZIP, Application, Basic, Documentation, Executable JAR, and Native Executable, as well as other types. Depending on your choice here, different defaults are set and different options are available as you work through the steps of the

wizard. For more information on other available archive types, choose the Help button in the wizard.



To begin creating an archive,

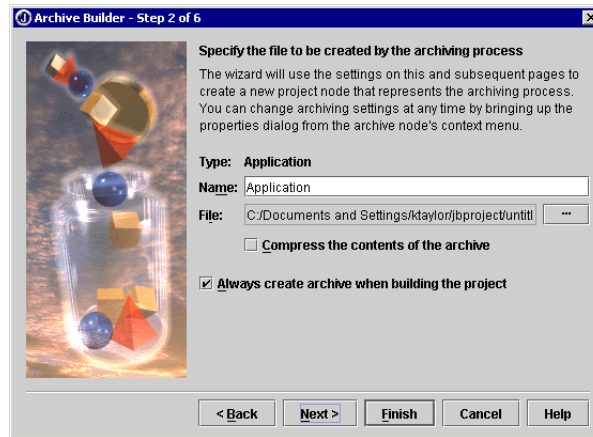
- 1 Build the project (Project | Make Project).
- 2 Choose Wizards | Archive Builder to open the Archive Builder. The Archive Builder is also available on the Build page of the object gallery (File | New | Build).
- 3 Choose an archive type from the Archive Type drop-down list. For more information on archive types, choose the Help button in the wizard.
- 4 Choose Next to continue to the next step of the wizard.

## Specifying the file to be created

The name of this step and the available options change according to the archive type selected. If the Web Start Application or Web Start Applet archive type, a feature of JBuilder Enterprise, is selected, the name of the step is Select Web Application. If the Executable JAR archive type is selected, the name of the step is Specify The Archive To Be Used.

In this step of the Archive Builder, you set a name for your archive node and file, select the archive's compression, and choose how frequently the archive is built. For documentation and source archives, this is the last step. If your archive type is Web Start Application or Web Start Applet, you also need to choose a web application (WebApp) defined in your project. To access a Web Start archive, the JAR must be in a web

application directory. If your archive type is Executable JAR, you need to choose the source archive for the executable that you want to create.



Although this step changes slightly according to the archive type selected, for most archive types, follow these basic steps:

- 1 Accept the name of the archive node in the Name field or enter a new name. The archive node displays in the project pane upon completion of the wizard, but the archive is not actually created until you make or rebuild the archive node. You can right-click the node and make or rebuild it at any time, as well as reset its properties. For more information on the archive node, see [“Understanding archive nodes” on page 15-31](#).
- 2 Enter the fully qualified path and file name for the archive to be generated by the Archive Builder. You can use the ellipsis (...) button to browse to a different directory location. For the Executable JAR archive type, choose an existing source archive for the executable that you want to create.

**Note** JAR files are only supported in browsers that support JDK 1.1 or later. If you are deploying an applet to a JDK 1.0.2 browser, you need to use a ZIP archive file.

- 3 Web Start Application or Web Start Applet archive types: choose a web application (WebApp) defined in your project.
- 4 Check or uncheck the Compress The Contents Of The Archive option. Usually, you leave this option off, so the archive is uncompressed and loading time is faster. If your archive is an applet, this option is selected by default. Because compression makes archive files smaller, applets are downloaded over the Internet or intranets faster when they’re compressed.
- 5 Check or uncheck the Always Create Archive When Building The Project option. This option determines how often your archive file is

created. This option is on by default for all archive types. When on, the archive file is recreated each time you make or rebuild your project.

- 6 Click Next to continue to the next step or click Finish if this is the last step. If this is the last step, generate the archive as described in [“Generating archive files” on page 15-31](#).

For more information on this step, choose the Help button in the wizard.

## Choosing deployment descriptor files

The J2EE Application Client archive type is a feature of JBuilder Enterprise

If the J2EE Application Client archive type is selected, this step of the Archive Builder is where you choose the standard (`application-client.xml`) and any proprietary deployment descriptor files to be included in the META-INF directory of the archive. The J2EE Application Client must have an `application-client.xml` file and may have other server-dependent files.



To specify the deployment descriptor files for a J2EE Application Client archive,

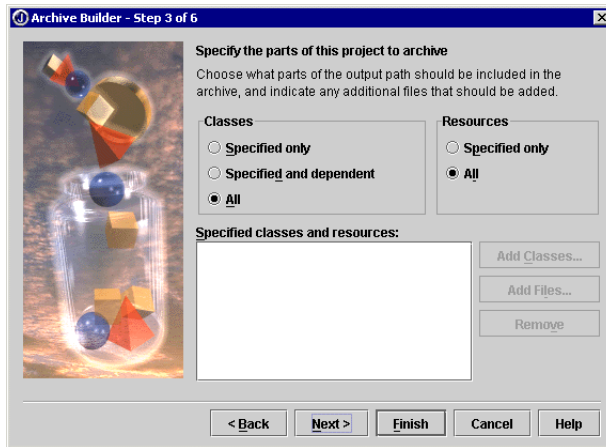
- 1 Choose the Create Descriptor(s) button and accept the default `application-client.xml` file name or enter another file name to create an empty deployment descriptor file. You can then edit this file before building the archive.
- 2 Choose the Add button to browse to any existing deployment descriptor file(s) to add to your archive.
- 3 Click Next to continue to the next step of the wizard.

For more information on this step, choose the Help button in the wizard.



## Specifying the parts of the project to archive

In this step of the Archive Builder, you choose what parts of the project are included in the archive. You can also choose additional classes or files.



To specify the parts of the project you want to include,

**1** Choose one of these options for classes:

- Specified Only
- Specified And Dependent
- All

For complete control over the classes included in the archive, choose Specified Only. If you choose Specified And Dependent, classes that you add with the Add Classes button are added to the archive, as well as any classes on the output path that the added classes depend on.

**Important**

If you choose Specified Only or Specified And Dependent, you must add the classes or packages with the Add Classes button.

**2** Choose one of these options for resources. If you choose Specified Only, you must add the resources with the Add Files button.

- Specified Only
- All

For example, if you want to include all the project classes and resources in the archive, you would select All for both classes and resources. If you don't want to include any dependencies in your archive and only specified resources, you would choose Specified Only for both classes and resources and then add the classes and resources you want with the Add Classes and Add Files buttons. To add classes and files, classes must be on the project output path and files must be on the project

source path. For more information on these options, choose the Help button in the wizard.

- 3 Choose the Add Classes button if you selected Specified Only or Specified And Dependent in the Classes category. Then select the classes or packages to add to your archive. The classes must be on the project output path. If you choose Specified And Dependent, the Archive Builder scans these added class files for additional class dependencies and includes the dependencies in the archive.
- 4 Choose the Add Files button if you selected Specified Only in the Resources category. Then select the files to add to your archive. The files must be on the project source path. Use this option to add miscellaneous files to your archive, such as resources (.gif, .jpg, and audio files), property files, or archived documentation (.html, readme.txt).

**Note**

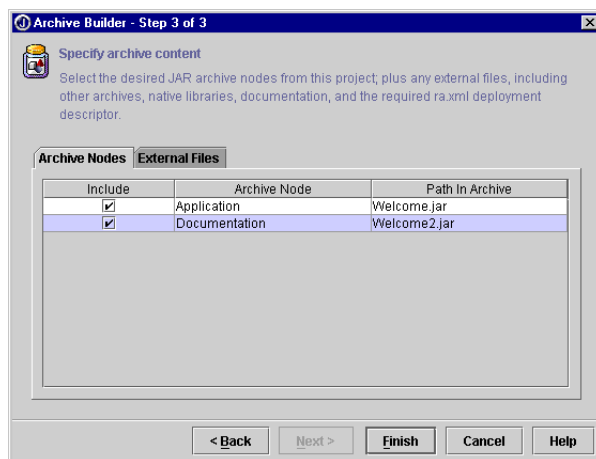
The Add Files dialog box can't look inside archive files. If a file or package you need is inside an archive file, extract it first to your source folder, then add it using the Add Files button.

- 5 Click Next to continue to the next step of the wizard.

## Specifying archive content for a Resource Adapter archive

The Resource Adapter (RAR) archive type is a feature of JBuilder Enterprise

If the Resource Adapter (RAR) archive type is selected, this step of the Archive Builder is where you choose JAR archive nodes from the project and any external files to add to your archive. External files can include other archives, libraries, documentation, and the required `ra.xml` deployment descriptor. The `ra.xml` file must already exist.



To specify the archive content for a Resource Adapter archive,

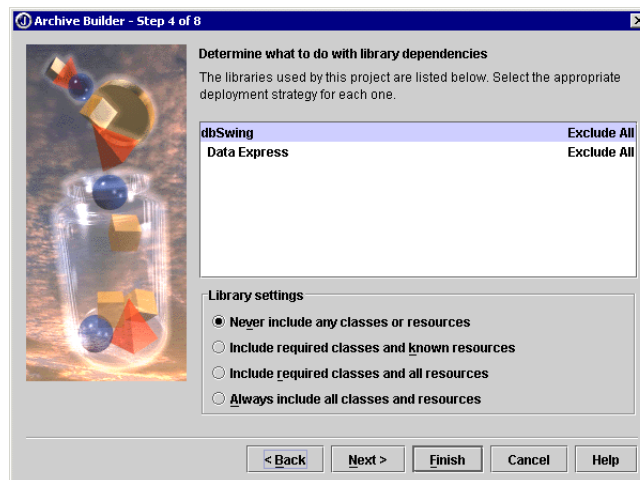
- 1 Check any JAR archive nodes on the Archive Nodes page that you want to include in the archive.
- 2 Choose any files to add to the archive, such as other archives, libraries, documentation, and the required `ra.xml` deployment descriptor. Choose the Add button to browse to a file.
- 3 Click Finish to close the wizard.
- 4 Generate the archive as described in [“Generating archive files” on page 15-31](#).

## Determining library dependencies

In this step of the Archive Builder, you determine what to do with library dependencies. The libraries used in your project are listed, and you can choose an individual deployment strategy for each one.

**Note** The Archive Builder never includes the JDK in your archive. It assumes that the JDK classes already exist on the target computer in the form of an installed JDK, Java runtime environment, or Java Plug-in, or that you are providing it in your installation.

**Important** In JDK 1.1.x, the Swing/JFC classes were **not** delivered as part of the JDK. If you are writing programs that use any of these JDKs, you must download and deliver `swingall.jar` which contains those files.



**Note** If you deploy any classes from the DataStore package (`com.borland.datastore`) or the VisiBroker package, you'll see a warning reminding you that deploying these packages requires a separate deployment license. If you already have the appropriate license and don't

want to see this warning again in this project, check “Don’t warn me about this project again.”

To specify library dependencies,

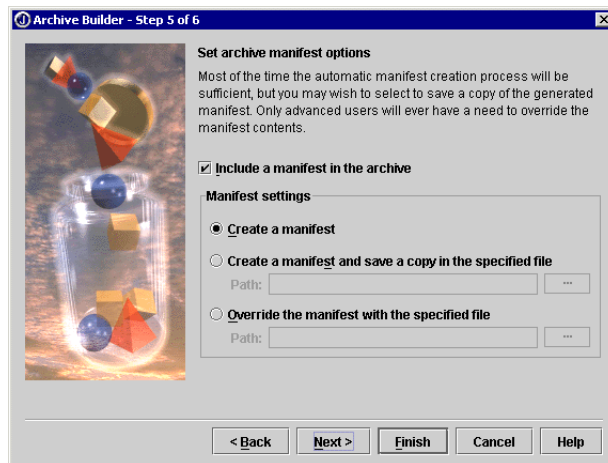
- 1 Select a library in the list.
- 2 Choose one of these options:
  - Never Include Any Classes Or Resources
  - Include Required Classes And Known Resources
  - Include Required Classes And All Resources
  - Always Include All Classes And Resources

Typically, Include Required Classes And All Resources is a good choice for library deployment. For more information on these options, choose the Help button in the wizard.

- 3 Select another library in the list and choose a library option.
- 4 Click Next to continue to the next step of the wizard.

## Setting archive manifest options

In this step of the Archive Builder, you choose how the manifest file is created. For most users, the default option, Create A Manifest, is sufficient. For more information on the manifest, see the topic called [“Understanding the manifest file” on page 15-3](#).



To set options for the archive manifest,

- 1 Accept the default, Include A Manifest In The Archive, if you want a manifest included.

**2** Choose one of these options if you want a manifest included:

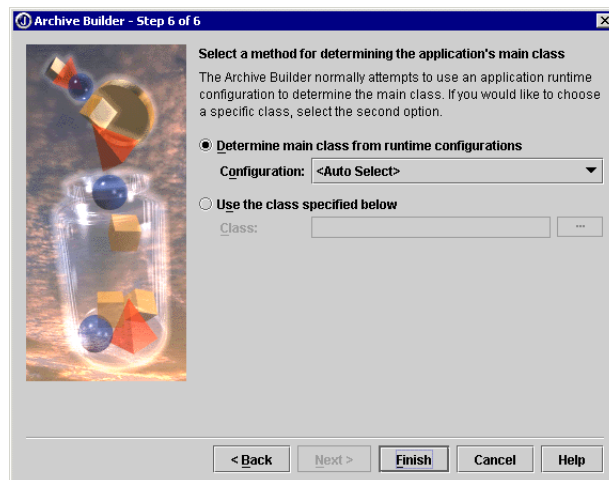
- Create A Manifest
- Create A Manifest And Save A Copy In The Specified File
- Override The Manifest With The Specified File

For more information on these options, choose the Help button in the wizard.

**3** Click Next to continue or click Finish if this is the last step of the wizard. If this is the last step, generate the archive as described in [“Generating archive files” on page 15-31](#).

## Selecting a method for determining the application’s main class

This step allows you to set the application’s main class. The main class runs the application. It contains the `public static void main(String[] args)` method.



To set the main class for the archive,

**1** Choose one of these options:

- Determine Main Class From Runtime Configurations

This option determines the main class from the selection in the Configuration drop-down list. The drop-down list includes all runtime configurations of type Application in the project. Choose a project runtime configuration or <Auto Select>. <Auto Select> uses the project runtime configuration marked as default. If there isn't a default or the default isn't an Application runtime configuration, the first Application runtime configuration in the project runtime

configuration list is used. For more about runtime configurations, see [“Setting runtime configurations” on page 7-6](#).

**Note**

If you’re creating a native executable or an executable JAR archive type and you specify a runtime configuration on this step of the wizard, the main class, application parameters, and VM parameters of this runtime configuration are included in the configuration file used to launch the executable. You can customize the configuration file on the last step of the wizard.

- Use The Class Specified Below

This option determines the main class from the specified class. Click the ellipsis (...) button and browse to and select a class.

- 2 Click Next to continue or click Finish if this is the last step of the wizard. If this is the last step, generate the archive as described in [“Generating archive files” on page 15-31](#).

For more information on this step, choose the Help button in the wizard.

**Caution**

If a main class isn’t specified, the following **won’t** execute:

- Launching native executables
- Using `java -jar <jarname>` from the command-line
- Double-clicking a JAR

## Determining which executable files to build

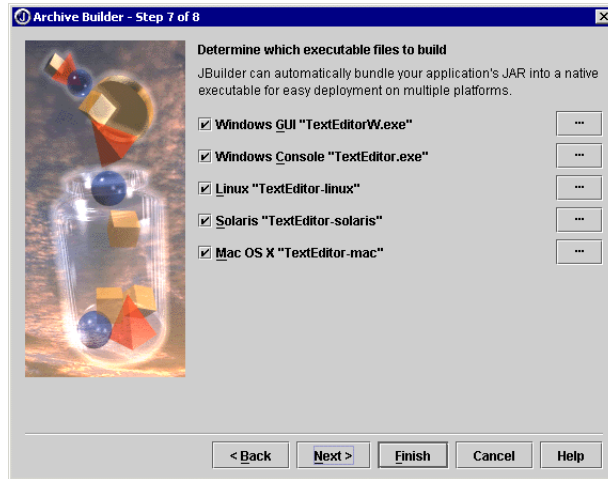
---

The Archive Builder can automatically bundle an application’s JAR file with native executable wrappers for easy deployment to various platforms.

This step is available when you select Native Executable or Executable JAR as the archive type in the Archive Builder and when you use the Native Executable Builder (Wizards | Native Executable Builder).

The executable file contains the executable launcher, the configuration file for the launcher, and the JAR file containing Java classes and resources in a single file. The Archive Builder sets the JAR comment to the configuration file and also adds the launcher executable to the beginning of the file. The Archive Builder allows you to customize the configuration file for the launcher on the last step of the wizard. Note that because the JDK is **not** bundled with the JAR file, it must be installed on the user’s computer to run the executable.

**Important** You must specify a main class on the previous step or the executable won't run.



To create an executable file,

- 1 Choose the executables you want to create. Click the ellipsis (...) button to rename and/or save the file to a different location.
- 2 Choose Next to set runtime configuration options for the executable.

For more information on this step, choose the Help button in the wizard.

## Running executables

Note that the JDK is **not** bundled with the JAR file, so the JDK must be installed on the user's computer for the executable to run. The platform-specific executable file looks for the installed JDK in the following location:

- Windows: Registry.
- Linux/Solaris: JAVA\_HOME environment variable and the user's path.
- Mac OS X: pre-defined location for the JDK.

**Note** You can override this default behavior by specifying the location of the JDK in a custom configuration file. Then the executable file will look in the specified location. For more information on configuration files, see the next step.

If you create the executable on the Windows platform and move it to other platforms, you may need to change the permissions to make it executable.

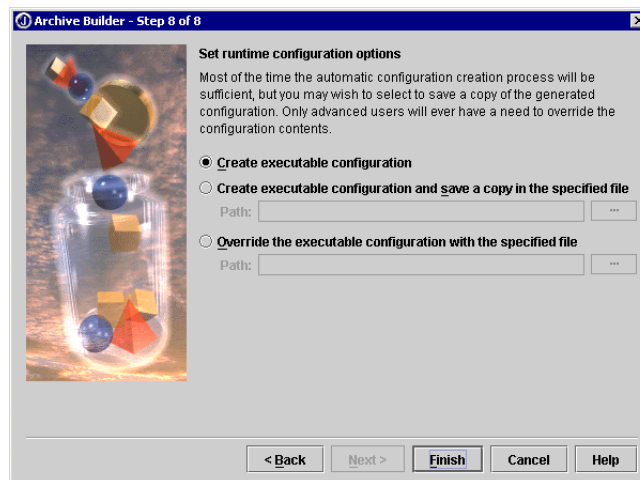
Choosing the Mac OS X option creates an application that is launchable only from a command line. To create an application that is launchable from the Finder, Mac users need to create an Application bundle. Please

refer to Apple's Mac OS X Developer documentation with regards to bundles and application packaging.

## Setting runtime configuration options

JBuilder automatically creates an executable configuration for launching the executable archive based on the runtime configuration specified on the step, *Selecting A Method For Determining The Application's Main Class*. If you choose a runtime configuration on that step of the wizard, the Archive Builder includes the main class, application parameters, and VM parameters in the configuration file that launches the executable. For more information on runtime configurations, see ["Setting runtime configurations" on page 7-6](#).

If you want to customize the executable configuration, you can modify the configuration that JBuilder creates or create your own configuration. For more information on creating configuration files, see [Appendix A, "Creating configuration files for native executables."](#)



To create an executable configuration for the archive,

**1** Choose one of these options:

- Create executable configuration
- Create executable configuration and save a copy in the specified file
- Override the executable configuration with the specified file

**Note**

If you choose to save a copy or override the configuration, the configuration file is added to the project.



- 2 Click Finish to close the wizard.
- 3 Generate the archive as described in [“Generating archive files” on page 15-31.](#)

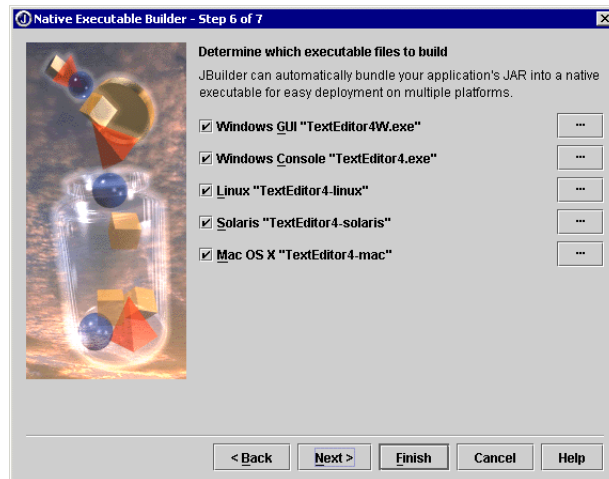
For more information on this step, choose the Help button in the wizard.

## Creating executables with the Native Executable Builder

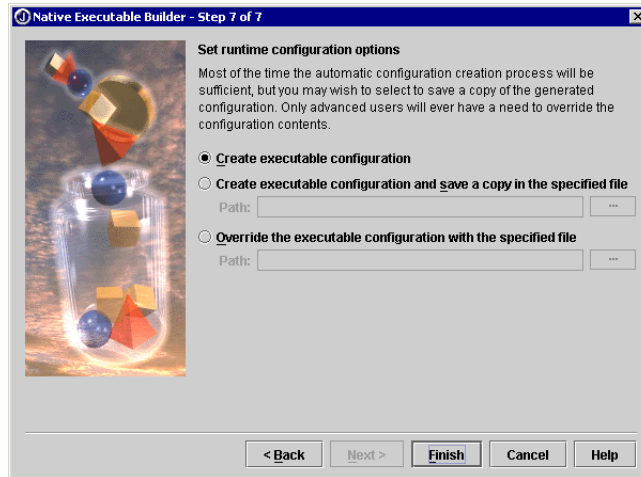
This is a feature of  
JBuilder SE and  
Enterprise

The Native Executable Builder automatically bundles an application JAR file with native executable wrappers for Windows, Linux, Solaris, and Mac OS X. Note that the JDK is **not** bundled with the JAR file, so the JDK must be installed on the user’s computer to run the executable. The Native Executable Builder, available on the Wizards menu and the Build page of the object gallery, is a shortcut to the Native Executable archive type of the Archive Builder. See [“Deploying with the Archive Builder” on page 15-17](#) for details about the Archive Builder.

JBuilder generates the selected executables and saves them with the project name and the appropriate file extension at the root of the current project directory. Choose the ellipsis (...) button next to the selection to change the default executable name and/or save it to a different location. Uncheck any executable you don’t want JBuilder to generate.



The Native Executable Builder also provides options for determining the main class from runtime configurations and creating custom configuration files to launch the executable.



The Native Executable Builder includes these steps, which are the same steps as the Executable JAR and Native Executable archive types of the Archive Builder:

- Specifying the file to be created
- Specifying the parts of the project to archive
- Determining library dependencies
- Setting archive manifest options
- Selecting a method for determining the application's main class
- Determining which executable files to build
- Setting runtime configuration options

**Important** Native executables **must** have a main class specified to execute.

Once you've completed the wizard, right-click the native executable node in the project pane and choose Make to create the executables and the JAR file. Expand the node to see the generated JAR file and executables. To modify the properties for this node, right-click and choose Properties.

### See also

- [Appendix A, "Creating configuration files for native executables"](#)
- ["Understanding archive nodes" on page 15-31](#)

## Generating archive files

---

When you exit the Archive Builder and the Native Executable Builder, an archive node is automatically displayed in the project pane. However, the archive file is not generated until you build the archive node.

When the archive node gets built is determined by an option set on the Specify The File To Be Created step of the Archive Builder and the Native Executable Builder: the Always Create Archive When Building Project option.

- If this option is on, the archive file is built each time you choose Project | Make Project or Project | Rebuild Project.
- If this option is off, you can create the archive file by right-clicking the archive node in the project pane and choosing Make or Rebuild.

## Understanding archive nodes

---

Using the Archive Builder or Native Executable Builder, you can create several archive files with different settings to test various deployment scenarios. First, use the Archive Builder to include different classes, dependencies, and resources in various combinations. Then, by comparing the contents of each archive file, you'll discover the strategy that best meets your size, download time, and installation requirements.

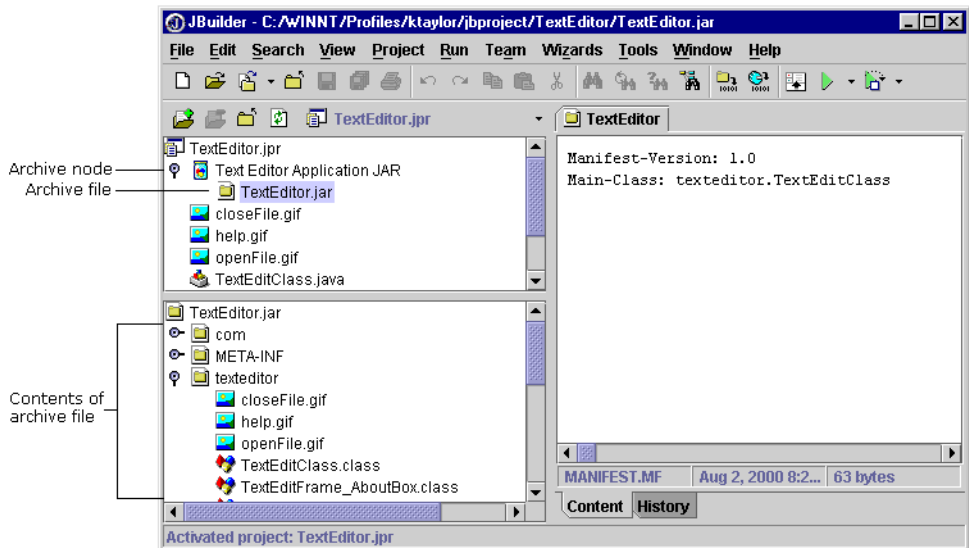
At any time during development, you can make the archive file, rebuild it, or reset its properties. You can also view the contents of the archive, as well as the contents of the manifest file.

### Viewing the archive and manifest

---

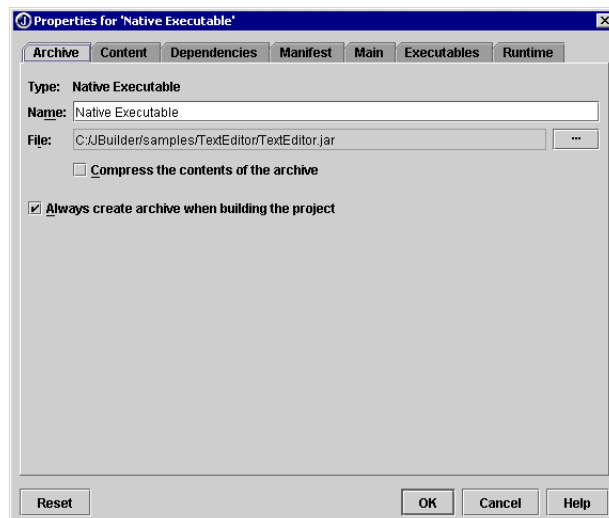
To view the archive file and the manifest file, expand the archive node in the project pane. Double-click the archive file in the project pane to display its contents in the structure pane and the manifest file in the

content pane. Double-click other files in the structure pane to open them as read-only files in the editor.



## Modifying archive node properties

At any time during development, you can reset the archive node's properties to change the contents of the resulting archive file. To change properties, right-click the archive node and choose Properties. The Properties dialog box displays pages that correspond to the steps of the Archive Builder and the Native Executable Builder.



## Removing, deleting, and renaming archives

---

After creating several archive versions for your project, you may want to remove, delete, or rename an archive that you no longer want. There are several ways to do this, depending on what you want to do.

Removing the archive node does not actually delete the JAR file, but it does remove the JAR file and the archive node from the project. You can always add the removed archive to your project again later if you want to.

To remove the archive node and its contents from your project, do one of the following:

- Right-click the archive node in the project pane and select Remove From Project.
- Select the archive node in the project pane and click the Remove From Project button on the project pane toolbar.
- Select the archive node in the project pane and choose Project | Remove From Project.

You can also delete the archive file from your project. This is useful if you want to reset the archive node properties and create a new archive file for the project. But keep in mind that if the Always Create Archive When Building The Project option is selected, the archive file is created again during the next project build. To see if this option is set, right-click the archive node and select Properties. This option, located on the Archive page, is on by default. After deleting the archive file, reset the archive properties, right-click the archive node, and select Make to recreate the archive with the new settings.

To delete the archive file, do one of the following:

- Right-click the archive node in the project pane and select Clean.
- Expand the archive node, right-click the JAR file, and select Delete <filename.jar>.

Lastly, you can rename archive nodes and files.

To rename archive nodes and files,

- Right-click the archive node or archive file in the project pane and choose Rename.
- Select the archive node or archive file in the project pane and choose Project | Rename.



# Internationalizing programs with JBuilder

This is a feature of  
JBuilder SE and  
Enterprise

This section examines issues involved in designing your Java applications to meet the needs of a worldwide audience. Why limit the use of your applet or application only to users in a particular country, when with a little extra effort it could be used by people all around the world? Special features in JBuilder make it easy to take advantage of Java's internationalization capabilities, allowing your applications to be customized for any number of countries or languages without requiring cumbersome changes to the code.

Although this chapter is about specific JBuilder features and is not meant to be an in-depth discussion of Java's internationalization features, several links are provided to related Java documentation to help get you started. Before proceeding to the explanation of [“Internationalization features in JBuilder” on page 16-2](#), please review the following section on commonly-used terms that are specific to internationalization.

## Internationalization terms and definitions

---

- **Internationalization (i18n)**

Internationalization is the process of designing or converting an existing program so it is capable of being used in more than one locale. Because internationalization is a long word, it is often abbreviated as ‘i18n’, where 18 represents the number of letters between the ‘i’ and ‘n.’

- **Locale**

A locale defines a set of culturally-specific conventions for the display, format, and collation (sorting) of data. In Java, a locale is specified by a `Locale` object, which is simply a container for strings identifying a particular language and country.

- **Resourcing**

Resourcing is the part of the internationalization process that involves isolating the locale-specific resources in the source code into modules so they can be independently added to or removed from the application. Examples of locale-specific resources include text displayed to the user or possibly even business rules or application logic. Java provides a set of `ResourceBundle` classes for resourcing strings and objects in Java programs.

- **Localization (l10n)**

Localization is the customization of a program's resources for a particular locale. Note that whereas internationalization generalizes a program for any locale, localization specializes it for a single locale. Because localization is a long word, it is often abbreviated as 'l10n', where 10 represents the number of letters between the 'l' and 'n.'

- **Native encoding**

A native encoding, also commonly known as a character set or codepage, defines a mapping of numeric values to symbolic characters within a particular operating system. Because the native encoding varies by operating system (and sometimes even within the same operating system), a file containing characters on one system may appear to have completely different characters on another system using a different native encoding.

- **Unicode**

Unicode is a universal character encoding standard maintained by The Unicode Consortium that defines a character mapping for nearly all the written languages of the world. Any Unicode character can be specified in Java source code by its Unicode escape sequence, `\uNNNN`, where `NNNN` is the hexadecimal value of the character in the Unicode character set. Characters and strings are always processed as 16-bit Unicode-encoded values within the Java Virtual Machine.

## Internationalization features in JBuilder

---

JBuilder includes a number of features designed to help you easily internationalize your Java applets and applications. The following sections discuss these features:



- A multilingual sample application
- The Resource Strings wizard, which is used to eliminate hard-coded strings
- dbSwing internationalization features
- Locale-sensitive components
- Components that display Unicode characters
- Internationalization features in the UI designer
- Unicode support in the IDE debugger
- Support for all JDK native encodings

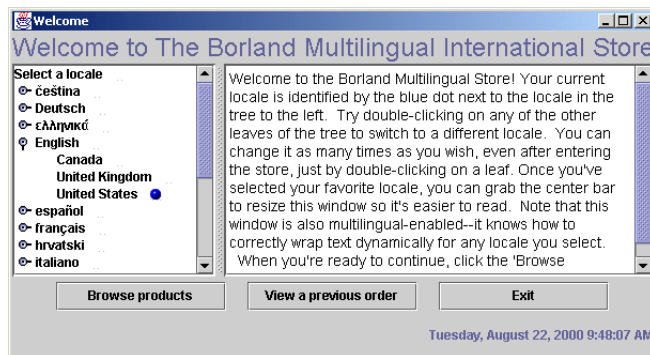
## A multilingual sample application

---

JBuilder includes an extensive multilingual sample order entry application demonstrating many of the important internationalization concepts in detail. This sample also illustrates many other important features of JBuilder, such as building applications with JBuilder components, creating internationalized JavaBeans, and using the DataExpress architecture.

You can find the `IntlDemo.jpx` project located under the `samples/dbswing/MultiLingual` directory of your JBuilder installation. Please refer to the `IntlDemo.html` documentation file and source code in the sample for more detailed information. The `IntlDemo` sample supports and includes translations for 15 different locales.

The Borland Multilingual International Store's `LocaleChooser` JavaBean lets you switch the application's locale at runtime. Doing so automatically adapts the UI to the language and conventions for the selected locale.

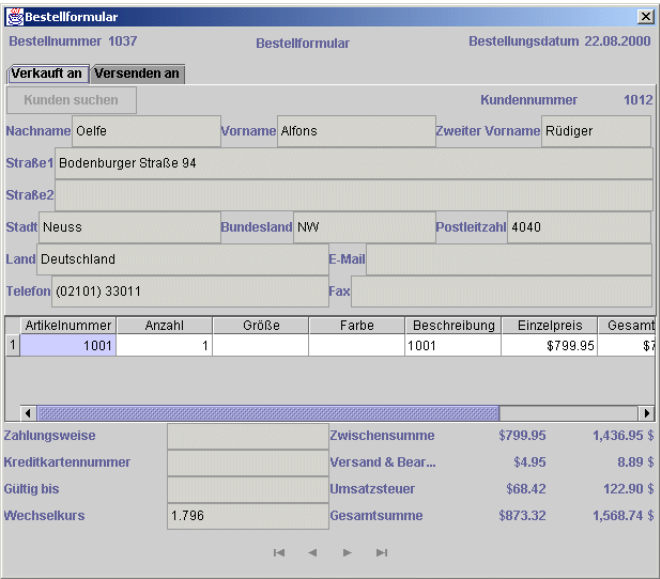


The `ProductFrame` lets users see images of Borland Store products and written descriptions in their own language. Note how the buttons and

labels adjust their sizes automatically for the different German and Italian translations shown here.



The OrderFrame displays the address of the customer and the cost of the order in the appropriate format for the user's locale. The OrderFrame is shown in German here:



## Eliminating hard-coded strings

---

A common design error that prevents your application or applet from being localized easily is the inclusion of hard-coded strings in your source code that are displayed in the UI of your application or applet.

While you can resource hard-coded strings in your user interface after you've completed and tested your source code, it's better to resource visible strings as part of the UI design process.

Resourcing your UI as you write it provides two major advantages:

- You don't have to go back and examine all the hard-coded strings in your source code and check which ones need to be resourced. Not only is this process very time-consuming, but sometimes it is difficult for you (or a colleague who is less familiar with your code) to determine which strings need resourcing.
- Resourcing strings early can help you discover non-internationalized UI designs earlier in your development process, saving you the effort of having to rewrite them later.

JBuilder provides two ways to get these benefits with minimal effort: the Resource Strings wizard and the Localizable Property Setting dialog box.

### Using the Resource Strings wizard

---

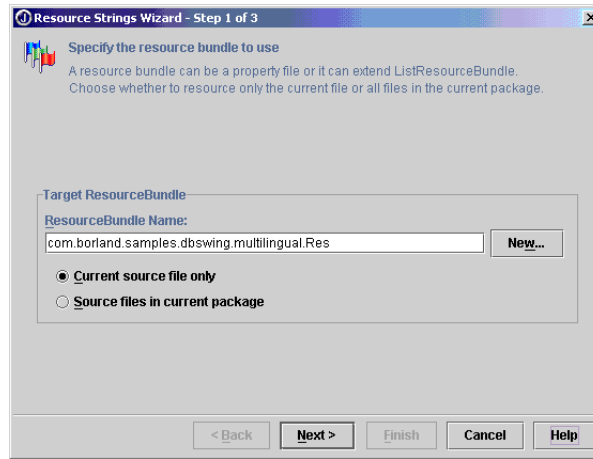
The Resource Strings wizard scans your source code and allows you to quickly and easily move hard-coded strings and single characters such as mnemonics into Java `ResourceBundle` classes. This wizard works with any Java file, not just source code generated by JBuilder.

`ResourceBundles` are specialized files that contain a collection of translatable strings. (They may also contain other types of data, though this is less common.) A unique resource key identifies each translatable string in the `ResourceBundle`. The hardcoded string in your application is replaced by a reference to the `ResourceBundle` and the resource key. This separation of application logic and translatable elements is called *resourcing*. These separate resource files are then sent to translators.

To move your strings into a `ResourceBundle` class,

- 1 In the project pane, double-click the source code file you want scanned to open it in the editor.

- 2 Choose Wizards | Resource Strings to display the Resource Strings wizard:

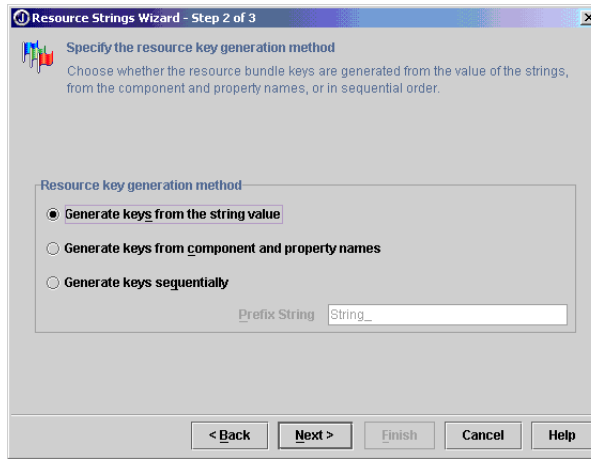


- 3 Specify the name of the ResourceBundle you are using. JBuilder suggests a default name. You can accept it or change it. By default, the ResourceBundle you create will be a ListResourceBundle. If you want to create a PropertyResourceBundle instead, click the New button and select PropertyResourceBundle from the Type drop-down list in the dialog box that appears and click OK.

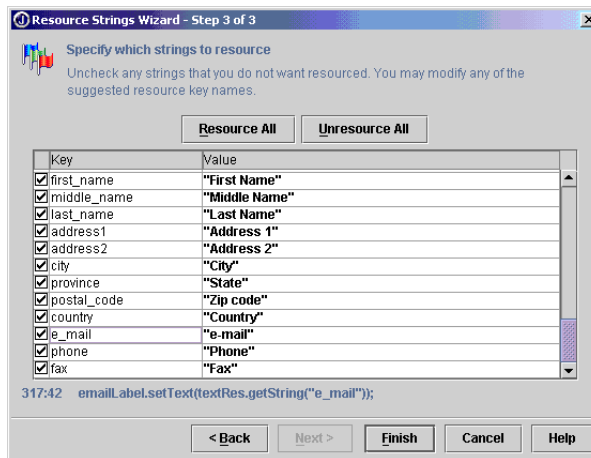
PropertyResourceBundles are text files with a `.properties` extension, and are placed in the same location as the class files for the source code. ListResourceBundles are provided as Java source files. Because they are implemented as Java source code, new and modified ListResourceBundles need to be recompiled for deployment. With PropertyResourceBundles, there is no need for recompilation when translations are modified or added to the application. ListResourceBundles provide considerably better performance than PropertyResourceBundles.

- 4 If you want just the current file in the editor resourced, select the Current Source File Only. If you want all files in the same package as

the file you have open to be resourced, select the Source Files In Current Package option, and click Next.



- 5 Specify how you want the keys generated. Each string is identified by a key. For example, if the string to be resourced is “Open File” and you select the Generate Keys From The String Value option, the key becomes `Open_File`. The same string might become `jbutton1_ToolTipText` if you select the Generate Keys From Component And Property Names. If you select the Generate Keys Sequentially option, the key might become `String_10`, if it’s the tenth string in the file. If you select the third option, you can also add a Prefix String that becomes the prefix of the name of the key. The default prefix is `String_`.
- 6 Choose Next to display the final page of the Resource Strings wizard:



To sort either the Key or Value column, click the Key or Value column header. Clicking the column header again displays the column in reverse order.

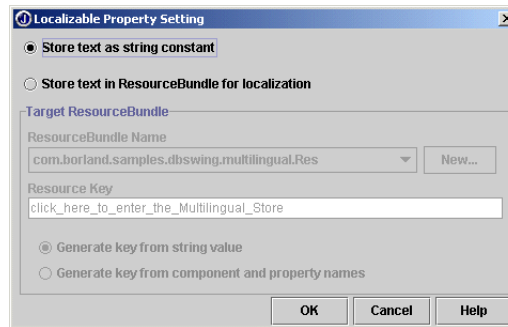
If you click a string displayed in the Resource Strings wizard, the line where the string appears is highlighted in your source code. You can use this feature to examine the string's context in your code.

- 7 Uncheck any string you don't want resourced.
- 8 Click Finish.

## Using the Localizable Property Setting dialog box

---

The Localizable Property Setting dialog box allows you to resource visible strings as you create or customize components in your UI. In the Inspector, simply right-click any text property, such as the label of a `ButtonControl`, and select the `ResourceBundle` command to display the Localizable Property Setting dialog box:



This dialog box displays options similar to those in the Resource Strings wizard but includes only those options that affect the single (selected) property. Select the `Store Text In ResourceBundle For Localization` option and select how the keys are generated if these options aren't already set or if you wish to make changes. Because resourcing from the Inspector is so quick and convenient, you can easily make it an integral part of customizing the components in your application.

## dbSwing internationalization features

---

dbSwing is a feature of  
JBuilder Enterprise

The dbSwing architecture includes several design decisions that facilitate internationalization of an application or applet:

- Non-internationalized, deprecated functions used in JDK 1.0 are avoided within dbSwing.
- All messages in dbSwing are stored in `ResourceBundle` classes, allowing applications built with these components to display text in the correct language for the end user's locale.

- All `dbSwing` components handle international issues such as locale-sensitive data formatting and locale-dependent collation.

Most JBuilder `dbSwing` components include a `textWithMnemonic` property. This property supports a mnemonic character that is specified in the same string used to display the component's text. The Swing design (which many `dbSwing` components extend) is to store the text and mnemonic characters into separate properties. This makes localization difficult as translators often have little context on which to base the translating of strings. Allowing `dbSwing` components to store the mnemonic character embedded in the text itself allows the translator to choose the correct mnemonic. If this feature is used effectively, it can provide some context during translation.

JBuilder's `IntlSwingSupport` component provides Swing internationalization support for twelve locales. When `IntlSwingSupport` is instantiated, it automatically updates Swing's internal localizable resources appropriately for the current locale. `IntlSwingSupport` need be instantiated once only in an application, and it should be instantiated on application startup before any Swing components are displayed.

To initialize `IntlSwingSupport` for a locale other than the default locale (in a multilingual application, for example), set the `locale` property of the `IntlSwingSupport` component to the target locale. For example:

```
new IntlSwingSupport();
int response = JOptionPane.showConfirmDialog(frame, localizedMessageString,
    localizedTitleString, JOptionPane.OK_CANCEL_OPTION);
```

**Note** `IntlSwingSupport` is meant to provide international support for some of the Swing components, not for `dbSwing` components. All `dbSwing` components are already fully internationalized.

As of JDK 1.2, the only Swing components with visible, translatable text strings are the `JFileChooser`, `JColorChooser`, and `JOptionPane`. For more information on locales, see the Sun documentation for the `Locale` class.

## Using JBuilder's locale-sensitive components

---

In addition to being fully resourced, many of JBuilder's components also provide useful locale-sensitive behavior. For example, string data that is loaded into a `Column` of a `JdbTable` using `DataExpress DataSet` components is automatically sorted according to the default collation order for the user's runtime locale. Similarly, date, time, and numeric values are automatically formatted correctly for the user's locale.

By default, objects inherit the locale of their containers. Therefore the locale setting on a `DataSet` is used by default by `Columns` within the `DataSet`. Alternatively, a `locale` can be specified explicitly for each `Column` object within the `DataSet`. This is useful if, for example, each `Column` holds data

that must be sorted by a different locale. Refer to the JDK's API documentation about the `Collator` class for more information about locale-sensitive sorting.

For more information about the locale-sensitive formatting of data types in Java, refer to the `DateFormat`, `NumberFormat`, and `MessageFormat` classes in the JDK API documentation.

## JBuilder components display any Unicode character

---

Component architectures which rely solely upon native UI peer controls to display characters can only display the set of characters supported by the native peer. Because JBuilder components use Java to display characters rather than native peers, they can display any Unicode character for which a font has been installed on your system, regardless of whether that character actually exists in your operating system's default character set.

To display Unicode characters for a new font,

- 1 Install the desired font on your operating system.
- 2 Modify the JDK `font.properties` file for your locale, specifying that the font for that character is now available.

For instructions on how to do this, refer to "Adding Fonts" in the JDK Internationalization documentation.

## Internationalization features in the UI designer

---

JBuilder's UI designer is a powerful tool for the creation and verification of your internationalized UI design. As you add translatable text elements to your UI, you can instantly put them into resource bundles. The Inspector automatically reads strings from and writes them back to resource bundles for you. In addition, after you've resourced all the text of your UI and have received a localized resource bundle from your translator, you can use the designer to quickly build and verify your internationalized user interface.

The Inspector displays locale-sensitive short description information about a JavaBean's property, as described in the internationalization section of the JavaBeans specification.

The Inspector allows the use of Unicode character escape sequences to denote characters that cannot be entered directly via the keyboard under your operating system locale. When you want to insert a Unicode character into a string property you're editing, simply put the hexadecimal value of the character's Unicode escape sequence within angle brackets. For



example, to insert the Japanese character for the word “mountain” into the label of a button, enter “<5C71>”. If your system has Japanese fonts installed and the proper settings in your JDK `font.properties` file, the character will be displayed as the label of the button, and the Unicode escape “\u5C71” will appear in your source code.

The UI designer provides excellent support for dynamic layout managers, a crucial requirement for building internationalized UI designs. Building a single UI capable of supporting multiple languages is a difficult task but one that is made much easier by the UI designer’s support for Java’s dynamic AWT layout managers. When designing a UI intended to be localized for more than one language, an extremely important rule is *always use a dynamic layout manager*.

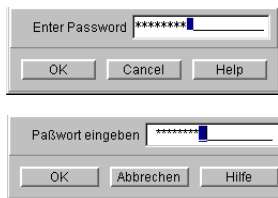
Consider, for example, the following `Dialog` containing OK, Cancel, and Help buttons:



This displays as expected for English labels, but when the labels are translated into German, the label’s text is too long to fit completely within the fixed button size. This is a very common problem that almost always occurs when attempting to localize a non-internationalized UI.



The solution is to use one or more dynamic AWT layout managers to allow the buttons to grow based on their label width. Here are the English and German internationalized versions of the same `Dialog`, written using a panel with a dynamic `GridLayout` for the buttons and embedded within a `BorderLayout` `Dialog`.



To learn more about creating dynamic layouts using the UI designer, refer to “Using layout managers” in *Designing Applications with JBuilder*.

The multilingual international sample application also demonstrates some advanced techniques for updating the layout of `Frames` in an application at runtime.

## Unicode in the IDE debugger

---

The JBuilder debugger allows you to view and edit Unicode characters, even if your operating system does not support them. When examining values in the debugger's Watch pane, expand the value in the tree you want to inspect until you can see its primitive Java type. By default, the debugger tries to display the Unicode character, assuming that your operating system can display it.

To view the character's Unicode equivalent, right-click the value and select the Show Hex Value option to see the character's Unicode escape sequence. You can also change the value by selecting Modify Value and entering another Unicode escape sequence in the Change Data Value dialog box.

## Specifying a native encoding for the compiler

---

The JBuilder and `javac` compilers compile source code encoded in native encodings (also known as `local codepages`), which is the storage format used by most text editors, including the JBuilder editor.

The IDE and compiler support all JDK native encodings. All JBuilder compilers automatically select the appropriate native encoding for your operating system's locale. You can also specify any JDK encoding for compiling source code files which were written in a different native encoding.

You can specify an encoding name to control how the compiler interprets characters beyond the English (ASCII) character set. The specification can be done on a project-wide basis or with the encoding compiler switch from the command line. If no setting is specified for this option, the default native encoding converter for the platform is used.

### Setting the encoding option

---

To set the encoding option from within the IDE,

- 1 Choose Project | Project Properties to display the Project Properties dialog box.
- 2 Click the General tab.
- 3 Select an encoding name from the Encoding drop-down list.
- 4 Choose OK.

To set the encoding option at the command line,

- 1 Use **bcj's -encoding** option followed by the encoding name.
- 2 Use **bmj's -encoding** option followed by the encoding name.

## Native encodings supported

---

Two encoding names have special meaning:

- **null**

Specifies that no native-encoding conversion should be done. Each byte in the file is converted to Unicode by setting it to the lower byte of the Unicode character. The upper byte of the Unicode character is set to zero.

- **default**

Equivalent to not specifying an encoding option. This uses the default encoding of the user's environment.

For a description of each encoding, see the JDK Internationalization Specification: Character Set Conversion: Supported Encodings at <http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html>. The following descriptions supplement that section:

- **Unicode**

Unicode, with BigEndian or LittleEndian indicated by Byte-Order-Mark.

- **UnicodeBig**

Big-Endian Unicode.

- **UnicodeLittle**

Little-Endian Unicode.

## Adding and overriding encodings

---

To add encodings to the Encoding drop-down list on the Build page of the Project Properties dialog box (Project | Project Properties),

- 1 Open the `user.properties` file in the `<.jbuilder>` folder.
- 2 Add encodings as in the following example:

```
compiler.java;encodings.add.1=ISO8859_2
compiler.java;encodings.add.2=ISO8859_3
```

- 3 Save and close the file.
- 4 Restart JBuilder.

To replace encodings in the Encodings drop-down list,

- 1 Open the `user.properties` file in the `<.jbuilder>` folder.
- 2 Override the encodings as in the following example:

```
compiler.java;encodings.override.1=ISO8859_2  
compiler.java;encodings.override.2=ISO8859_3
```

- 3 Save and close the file.
- 4 Restart JBuilder.

## More about native encodings

---

Non-Unicode environments represent characters using different encoding systems. In the PC world, these are known as `codepages`; Java refers to them as `native encodings`. When moving data from one encoding system to another, conversion needs to be done. Because each system can have a different set of extended characters, conversion is required to prevent loss of data.

You can also use the Java utility, **`native2ascii`**, to convert native-encoded characters to Unicode escape sequences (for example, `\uNNNN`). The converted file(s) can then be readily compiled on any system without the need for more conversion or the specifying of a particular encoding.

Most text editors, including JBuilder's editor, write text in the native encoding. For example, Japanese Windows uses the Shift-JIS format, and US Windows uses Windows Codepage 1252. Starting with JDK 1.1, **`javac`** is also able to compile "native-encoded" source code. The encoding can be specified by using the "encoding" switch. When the encoding is not specified, the compiler uses the encoding based on the user's environment.

Unlike Unicode, source code written with native encoding is not directly portable to systems using other encodings. For example, if source code has been encoded in Shift-JIS (a Japanese encoding), and you are running the compiler in a US Windows environment, you must specify the Shift-JIS encoding for the compiler to read the source correctly.

## The 16-bit Unicode format

---

Unicode is a universal system of representing characters using 16-bit numbers. The 16-bit Unicode character set can be supported directly, or can be represented indirectly within the 7-bit ASCII character set, using the `\u` escape character followed by four hexadecimal digits.

When all major operating environments directly support Unicode, this will replace the established approach, which requires conversion between

different native encodings with conflicting character values. Java is one of the first environments to standardize on Unicode; Unicode is the internal character set of the Java environment.

## Unicode support using ASCII and ‘\u’

---

Currently, most Windows text editors, including JBuilder’s editor, store and process text as 7- or 8-bit characters, rather than 16-bit Unicode characters. The ASCII character set uses a 7-bit encoding that contains the 26 letters of the English alphabet and some symbols. Almost all native encodings have ASCII as a subset, and represent it in the same way: the first 127 characters of an encoding are the ASCII character set. The ASCII character set can be considered a subset of Unicode.

To enable users to specify Unicode characters in their source code without a Unicode-enabled editor, the Java specification allows the use of the \u “Unicode escape” in an ASCII file. This usage enables extended characters to be represented by a combination of ASCII characters. This way of representing Unicode uses 6 characters to represent each non-ASCII character. To enter an ordinary ASCII character, you press the character’s key on the keyboard, and to enter a non-ASCII character, you type in the Unicode escape sequence representing the character.

In this 7-bit representation of Unicode, each character beyond the ASCII character set is represented in the form \uNNNN, where NNNN are the 4 hex digits of the Unicode character. For example, the Unicode character “Latin Small Letter F with Hook”, a cursive ‘f’ which is represented in Unicode with the hexadecimal number 0192, can be entered by typing “\u0192”.

Unicode, in both the 16-bit and 7-bit forms, is in a universal format; source code in Unicode is directly portable to all platforms, in all languages.

## JBuilder around the world

---

JBuilder is available in several languages including English, German, French, Spanish, and Japanese. Localized versions usually include translated documentation, UI, and components. Localized versions of JBuilder are available for purchase from the Borland sales office in those countries. To find links to JBuilder international sites, see Borland Worldwide at <http://www.borland.com/bww/>.

## Online internationalization support

---

Visit the multi-lingual-apps newsgroup on the Borland web page at <news://forums.borland.com/borland.public.jbuilder.multi-lingual-apps>. This newsgroup is dedicated to JBuilder internationalization and multilingual issues and is actively monitored by our support engineers as well as R&D and QA engineers in the JBuilder internationalization group.

## Tutorial: Compiling, running, and debugging

This tutorial is a feature of  
JBuilder SE and  
Enterprise

This step-by-step tutorial shows you how to find and fix syntax errors, compiler errors, and runtime errors in a sample provided with JBuilder.

- Syntax errors are identified before you compile. Syntax errors occur in code that does not meet the syntactical requirements of the Java language.
- Compiler errors are errors generated by the compiler: the syntax may be correct, but the compiler cannot compile the code due to missing variables, missing classes, or incomplete statements. Note that the true cause of an error might occur one or more lines before or after the line number specified in the error message.
- Runtime errors occur when your program successfully compiles but gives runtime exceptions or hangs when you run it. Your program contains valid statements, but the statements cause errors when they're executed.

The tutorial uses the sample project that is provided in the `<jbuilder>/samples/Tutorials/DebugTutorial` folder. The sample is a simple mathematical calculator. The program contains introduced errors and will not compile and run as provided. You must work through this tutorial, finding and fixing the errors, in order for the program to run.

**Important** The line numbers in this tutorial may not match the line numbers displayed in JBuilder.

## Step 1: Opening the sample project

For more information on compiling, running, and debugging, read the following chapters:

- [Chapter 6, “Building Java programs”](#)
- [Chapter 5, “Compiling Java programs”](#)
- [Chapter 7, “Running Java programs”](#)
- [Chapter 8, “Debugging Java programs”](#)

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 1-4](#).

## Step 1: Opening the sample project

---

In this step, you will open the project file and open one of the files in the project. You’ll see that syntax errors exist in one of the files.

To open the sample project,

- 1 Choose File | Open Project. The Open Project dialog box is displayed.
- 2 Navigate to the `samples/Tutorials/DebugTutorial` folder of your JBuilder installation.
- 3 Double-click `DebugTutorial.jpx`. The project is opened in the project pane. The files in the project are listed in the project pane. This project consists of three files:
  - `Application1.java` - The runnable file, containing the `main()` method.
  - `DebugTutorial.html` - The HTML file that provides a descriptive overview of the project.
  - `Frame1.java` - The file that contains the frame, the components, and the methods for the program.
- 4 Double-click `Frame1.java`. This opens the file in the editor and displays its structure in the structure pane.

Notice the Errors folder in the structure pane. You will be finding and fixing these errors in Step 2 of the tutorial.



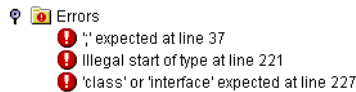
## Step 2: Fixing syntax errors

Syntax errors do not meet the syntactical requirements of the Java language. JBuilder identifies these errors before you compile. They are listed in the Errors folder of the structure pane. If you try to compile the program without fixing these syntax errors, JBuilder will display the errors in the message pane. The program cannot be compiled until these errors are fixed.

In this step, you will find the syntax errors in the sample program and fix them. For more information on JBuilder's error messages, see the online topic called "Compiler error messages."

To find and fix syntax errors,

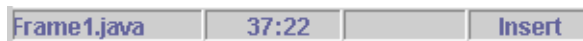
- 1 Expand the Errors folder in the structure pane.



Three errors are listed. The first error indicates that a semi-colon is missing from the end of the line of code.

- 2 Click the first error in the structure pane. JBuilder moves the cursor to the matching line of code in the editor. If you single-click the error, JBuilder highlights the matching line of the code. A double-click places the cursor in the column where the error occurred.

**Tip** The content pane's status bar displays the line and column number, as well as the insert mode.



- 3 Add a semi-colon to the end of the line. You've fixed the error, and it is removed from the structure pane.
- 4 Click the next error in the structure pane. JBuilder moves the cursor to the matching line of code in the editor. This error is a little trickier to decipher. The message means that a type identifier was expected at this point in the program, but was not found.

Notice that the line of code starts with the keyword `else`, and that the next line consists of a single closing brace. If you read the previous lines of code, you'll notice the beginning of an `if` statement. In Java, an `if` statement must include an opening and closing brace. However, if you look on the line with the `if` statement, you'll see that the opening brace is missing.

- 5 Add an opening brace to the end of that line. The completed line of code will look like this:

```
if (valueOneOddEven) {
```

Notice how the editor's brace matching feature shows you the matching closing brace.

The remaining two syntax errors are removed from the structure pane. Different errors are now displayed in the Errors folder. You will fix these errors in the next step.



- 6 Click the Save All button on the main toolbar.

Sometimes it takes a bit of detective work to correct syntax errors. Often, fixing one syntax error will fix several errors listed in the structure pane. In this case, for example, the third syntax error was: 'class' or 'interface' expected at line 227. Because the closing brace did not have a corresponding opening brace, JBuilder expected to find a class declaration after the close of the current method. However, when the opening brace was added, JBuilder could determine that the brace now had a match and that the next line of code was not in error.

**Tip** You can find matching braces by moving your cursor to the brace. The matching brace is highlighted.

In the next step, you'll find and fix errors that would prevent this program from compiling.

## Step 3: Fixing compiler errors

---

In this step of the tutorial, you will find and fix errors that would prevent your program from compiling. These errors are displayed in the Errors folder of the structure pane.

To find and fix compiler errors,

- 1 Click the first error in the Errors folder:

```
Constructor Double() not found in class java.lang.Double at line 38
```

JBuilder positions the cursor on the matching line of code:

```
Double valueOneDouble = new Double();
```

The error message indicates that the Java class `java.lang.Double` does not contain a parameterless constructor. The highlighted statement is attempting to create a new `Double` object that does not have a parameter. If you look at the constructors in the `java.lang.Double` class, you'll see that all constructors require a parameter. Additionally, if you look a

few lines further on in the program, you'll see that the `Double` object, `valueTwoDouble`, is constructed with an initial value of `0.0`.

**Tip** Position the cursor between the parenthesis and press *Ctrl+Shift+Space* to display ParameterInsight, JBuilder's pop-up window that displays the required parameter type. You can also right-click the `Double()` method and choose Find Definition to open the source in the editor.

- 2 Insert `0.0` between the parenthesis. The statement will now read:

```
Double valueOneDouble = new Double(0.0);
```

**Tip** The content pane status bar now displays the word `Modified`, indicating that you've made changes to the file.



- 3 Click the Save All button on the toolbar.

- 4 Click the next error in the Errors folder:

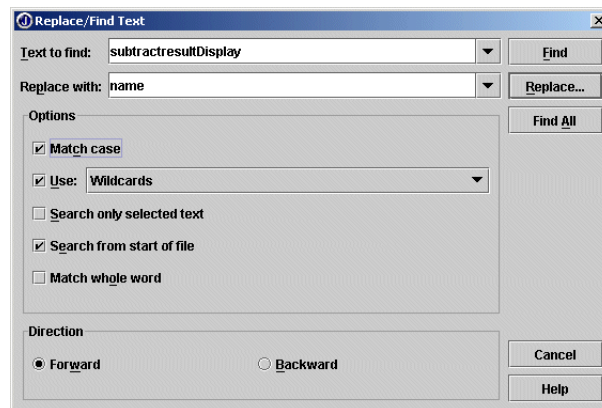
```
Variable subtractresultDisplay not found in class DebugTutorial.Frame1 at  
line 243
```

This error indicates that the variable `subtractresultDisplay` has not been defined.

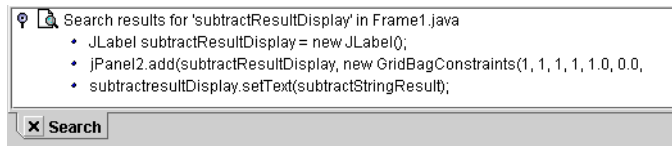
- 5 Choose Search | Find to display the Find/Replace Text dialog box.

**Tip** If the Find command is dimmed, click in the editor and choose Search | Find again.

- 6 Enter `subtractresultDisplay` in the Text To Find field. Make sure the Match Case option is turned off. Click the Search From Start Of File option to start the search from the beginning of the file.



- 7 Click Find All. The results of the search are displayed on the Search tab of the message pane.



Notice that two of the three references to this label are `subtractResultDisplay`; there is an uppercase R in Result. Casing is critical in Java: `subtractresultDisplay` is not the same as `subtractResultDisplay`.

- 8 Double-click the incorrect reference in the Search tab to move the cursor to the reference in the editor.
- 9 Change `subtractresultDisplay` to `subtractResultDisplay`.
- 10 Click the X on the Search tab to close it. (You can also right-click the tab and choose Remove "Search" Tab.)

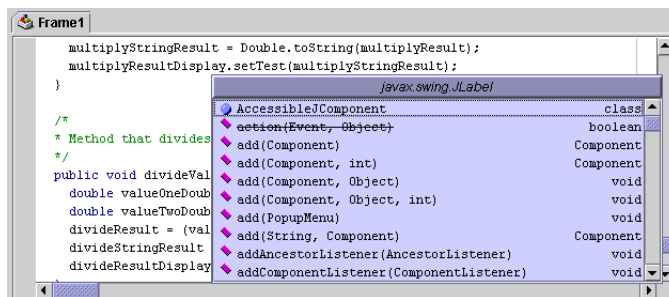
Use CodeInsight to fix the remaining error:

- 1 Click the remaining error in Errors folder. This error indicates that there is no `setTest()` method in `javax.swing.JLabel`.

```
Method setTest(java.lang.String) not found in class javax.swing.JLabel at
line 254
```

JBuilder positions the cursor on the matching line of code.

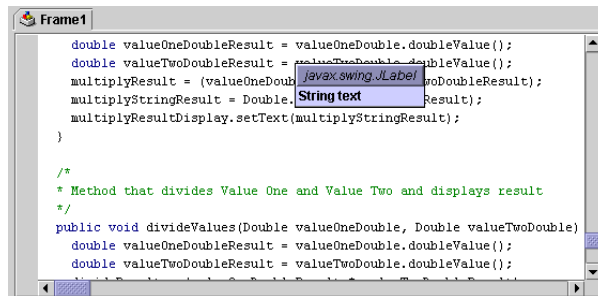
- 2 Position the cursor after the dot (.) and press **Ctrl+Space**. This displays the CodeInsight pop-up window that lists available member functions.



**Note**

If the pop-up window is not displayed, see “Keymaps for editor emulations” (Help | Keyboard Mappings) for a list of CodeInsight keystrokes.

- Scroll through the window using the arrow keys. Those items that are in bold-faced type are in this class. The items with lines through them have been deprecated. The grayed-out items are inherited, but are available for use.
- Search for `setText` by typing `setText` or scrolling. Once selected, double-click it or press *Enter*. The `setText()` method is inserted in the editor after the dot, replacing the incorrect `setTest` method name. A tool tip displays the expected method parameter type.



### 3 Click the Save All button on the toolbar.

In the next step, you'll examine the runtime configuration and run the program.

## Step 4: Running the program

In this step of the tutorial, you will examine the program's runtime configuration and run the program.

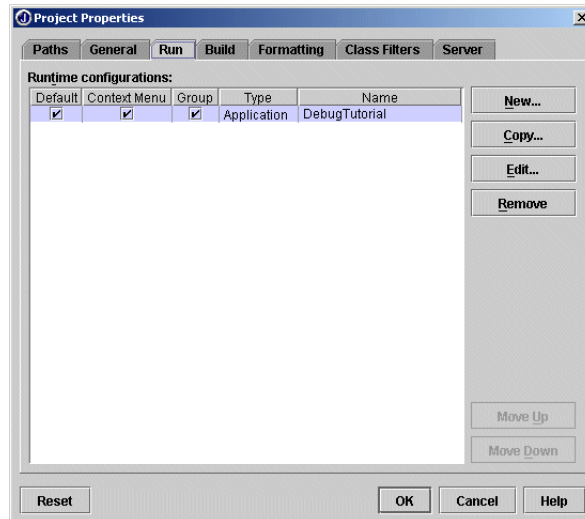
Runtime configurations are preset runtime parameters. Using preset parameters saves you time when running and debugging, because you only have to set the parameters once. With preset configurations, each time you run or debug you simply select the desired configuration.

For more information on runtime configurations, see [“Setting runtime configurations” on page 7-6](#).

## Step 4: Running the program

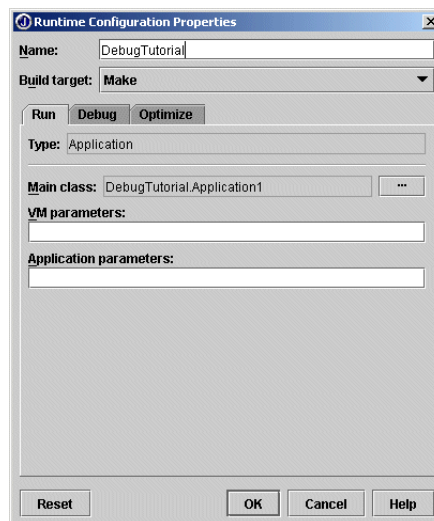
To examine the runtime configuration for this application,

- 1 Choose Run | Configurations. The Run page of the Project Properties dialog box is displayed.



There is one preset configuration - DebugTutorial. It is an Application configuration, meaning that an application runner is used for running. It is the default configuration, and will also be displayed on the context menu when you right-click `Application1.java`, the runnable file.

- 2 Click the Edit button. The Runtime Configuration Properties dialog box is displayed.



Notice that the Type is set to application; the application runner is used. The Main Class is set to `DebugTutorial.Application1`.

- 3 Click the Debug tab to view debug properties for the runtime configuration. The Debug page is displayed.



Notice that Smart Step is enabled.

- 4 Click OK two times to close the Runtime Configuration Properties dialog box and the Project Properties dialog box.

## Saving files and running the program

Save your changes and run the program:



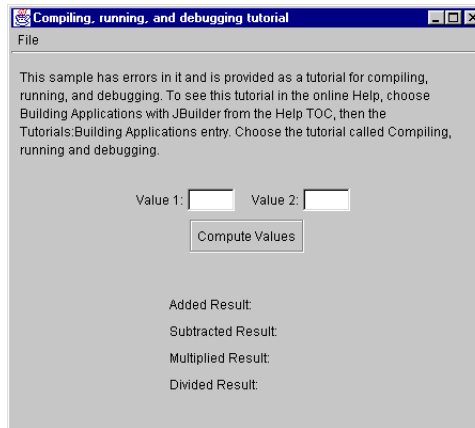
- 1 Click the Save All button on the toolbar.



- 2 Click the Run Project button on the toolbar. The program runs using the DebugTutorial configuration, the default configuration. Compiler

## Step 5: Fixing the `subtractValues()` method

output is displayed on the Application1 tab in the message pane. The program UI is displayed.



- 3 Enter whole numbers into the program's Value 1 and Value 2 input fields. Press the Compute Values button. The values are computed and displayed. However, if you look carefully at computed results, you'll see that there are some runtime errors; the program compiles and runs but gives incorrect results. You will find and fix these errors in the next steps.
  - 4 Choose File | Exit to exit the application.
  - 5 Click the X on the Application1 tab in the message pane to close it.
- In the next step, you'll find and fix a runtime error.

## Step 5: Fixing the `subtractValues()` method

---

In this step of the tutorial, you will find and fix one of three runtime errors. To find this error, you'll use debugger features. You'll learn how to:

- Start and stop the debugger.
- Create a floating window for one of the debugger views.
- Set a breakpoint.
- Step into and step over a method.
- Trace into a thread.
- Set a `this` watch, an object watch, and a local variable watch.
- Use the Evaluate/Modify dialog box.

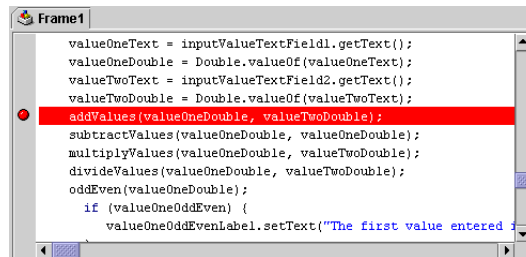
In the previous step, you ran the program. When you entered values into the Value 1 and Value 2 input fields, and pressed Compute Values to



compute the added, subtracted, multiplied, and divided values, you may have noticed that the subtracted value was not correct. For example, if you enter 4 in the Value 1 field and 3 in the Value 2 field, the subtracted result is 0.0 instead of 1.0.

To find this error, we'll use the debugger. First, we'll set a breakpoint and start the debugger.

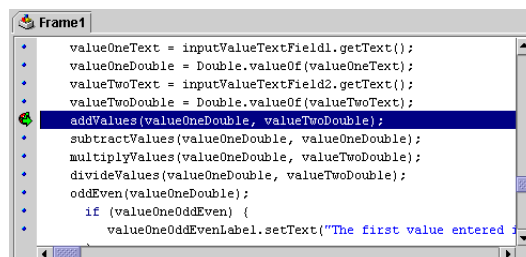
- 1 Use the Find/Replace Text dialog box (Search | Find) to find the line of code that calls the `addValues()` method. This is the first method called when the Compute Values button is pressed. Enter `addValues` in the Text To Find field of the dialog box to locate the call to the method. Press the Find button.
- 2 Click the gray gutter in the editor to the left of the line of code. A breakpoint is set on this line. The red circle icon indicates that the breakpoint has not been verified.



- 3 Click the Debug Program button on the toolbar. JBuilder starts the debugger VM, using the DebugTutorial runtime configuration.

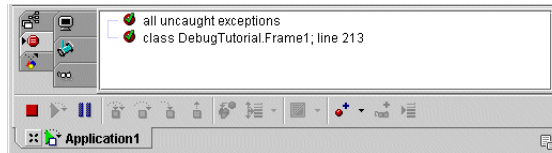
The program is now running and waiting for user input (this may take a few moments).

- 4 Enter 4 in the Value 1 field and 3 in the Value 2 field. Press Compute Values. Before you can examine the results, the debugger takes control. The program is minimized and the debugger is displayed in the message pane. Blue icons are now displayed in the editor next to executable lines of code, showing where valid breakpoints can be set. The icon for the breakpoint you just set has changed to a red dot with a green checkmark to show that the breakpoint is valid. The arrow indicates the execution point (in this case, the breakpointed line is also the execution point).



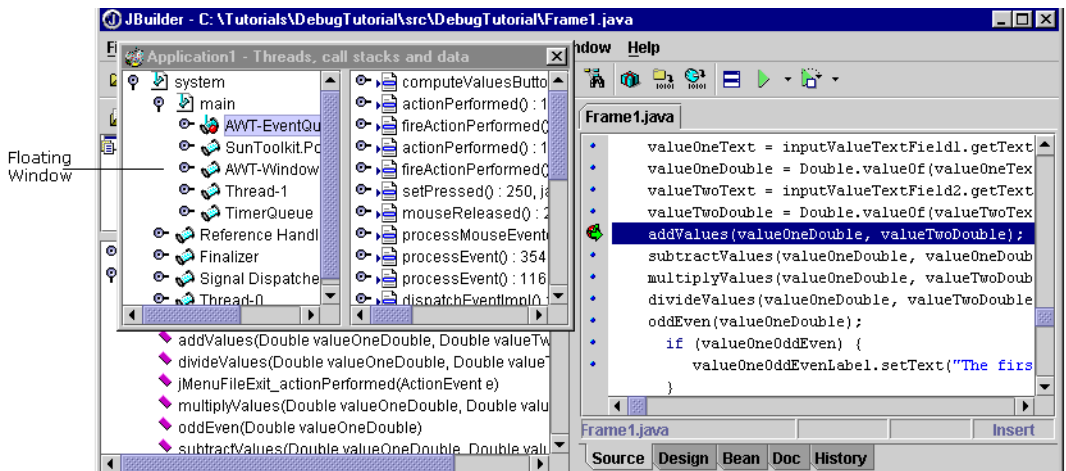
For information on the debugger UI, see [“The debugger user interface” on page 8-8](#).

- 5 Click the Breakpoints tab on the left side of the debugger to go to the Data and code breakpoints view. The default breakpoint and the breakpoint you just set are displayed. The debugger status bar displays a message indicating that the program has stopped on the breakpoint you set in the editor.



The next step is to trace into the stepping thread. This allows you to see where methods are called and set watches on those methods.

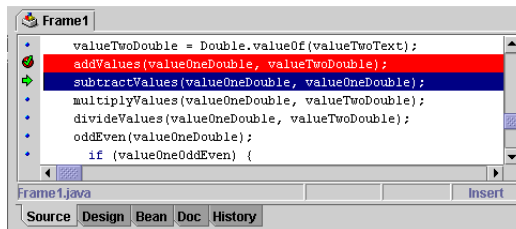
- 1 Go to the Threads, call stacks, and data view. Notice how the view is split, allowing you to see the contents of the item selected on the left pane on the right pane.
- 2 Right-click an empty area of the left pane of the view and choose Floating Window. The view now turns into a floating window and is initially displayed at the top left of the screen. You can resize the window or move it. Changing a view to a window allows you to look at more than one debugger view at a time. (Note that all views, except the Console, output, input and errors view, can be turned into floating windows.)



- 3 Scroll in the editor so that you can see the breakpoint and the floating window at the same time.
- 4 You can also set the breakpoint on the call to the subtractValues() method instead of the addValues() method, allowing you to get closer to



the actual area of the program you want to examine more closely. To do this, click the Step Over button on the debugger toolbar. This steps over the call to the `addValues()` method, positioning the execution point on the call to the `subtractValues()` method.

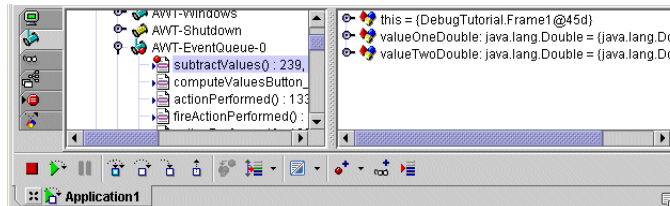


- 5 Click the Step Into button to step into the `subtractValues()` method. The `subtractValues()` method is now highlighted in the left pane of the floating Threads, call stacks and data view.

- 6 Right-click an empty area on the left pane of the floating Threads, call stacks and data view and uncheck Floating Window to close it. The floating window is displayed again as a debugger view.

**Tip** If you want to reset the debugger tabs to their default order, right-click an empty area of the view and choose Restore Default View Order.

- 7 Go to the Threads, call stacks, and data view. Notice that the `subtractValues()` method is expanded on the right pane of the view.



**Tip** You can use the Show Current Frame button to display the thread being stepped into.

The next step is to set watches on objects and variables. This allows you to examine data values.

- 1 Create a `this` object watch by right-clicking the `this` object in the expanded list:

```
this = {DebuggerTutorial.Frame1@3c6}
```

Choose the Create 'this' Watch command. A watch on the `this` object allows you to trace through the current instantiation of the class.

- 2 The Add Watch dialog box is displayed, with the Enter A Watch Description field available. Click OK.



You do not need to enter a description for the watch. If you do enter a description, it is displayed on the same line as the watched expression in the Data watches view. A description may make individual watches easier to locate in the view.

- 3 Right-click the `this` object again:

```
this = {DebuggerTutorial.Frame1@3c6}
```

This time, choose the Create Object Watch command to create an object watch. The Add Watch dialog box is displayed. Click OK.

- 4 Right-click the `valueOneDouble` object in the expanded list (on the right pane) to create a watch on the first value being passed to the `subtractValues()` method:

```
valueOneDouble: java.lang.Double. = {java.lang.Double@3c7}
```

Choose the Create Local Variable Watch command. The Add Watch dialog box is displayed. Click OK.

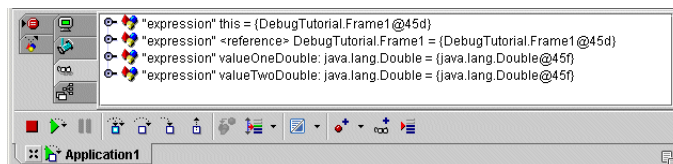
- 5 Right-click the `valueTwoDouble` object in the expanded list to create a watch on the second value being passed to the method:

```
valueTwoDouble: java.lang.Double. = {java.lang.Double@3c7}
```

Choose the Create Local Variable Watch command. The Add Watch dialog box is displayed. Click OK.



- 6 Go to the Data watches view.

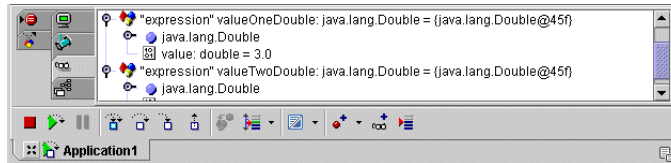


- 7 Expand the first two watches: the `this` watch and the `<reference>` watch. In this case, both the watches provide the same data, as the two watches are identical. Note that you can watch all object data in this view (except static data). The grayed-out items are inherited. Collapse these two watches. The remaining two watches, the local variable watches, watch the values of `valueOneDouble` and `valueTwoDouble`.



- 8 Click the Step Into button to step into the `subtractValues()` method.

## 9 Expand the watches on valueOneDouble and valueTwoDouble.



The two values are equal. You did not enter two equal values into the program's two input fields.

## 10 Set a watch on subtractStringResult, the result of the subtraction. This value, a String, is written to the output label. To set the watch, click the Add Watch button on the debugger toolbar, and enter subtractStringResult in the Expression field. Click OK. You may have to scroll the Data watches view to see the watch.



## 11 Click the Step Into button three times to step to the following line in the editor:



```
subtractResultDisplay.setText(subtractStringResult)
```

In the Data watches view, subtractStringResult is set to 0.0 instead of 1.0, as expected.

### Note

You could also use the Evaluate/Modify dialog box to examine the value of subtractStringResult. To do this, choose Run | Evaluate/Modify. Enter subtractStringResult into the Expression input field, and click Evaluate. The result of the evaluation is displayed in the Result field. Note that the display is similar to expanding the watch. Click Close to close the dialog box.

## 12 Step into the method two more times. The execution point returns to the line where the next method, multiplyValues(), is called.

## 13 Look at the call to the subtractValues() method, the line before the execution point. Notice that valueOneDouble is being passed twice, instead of valueOneDouble and valueTwoDouble. Change the second parameter to valueTwoDouble.

## Saving files and running the program

Save your changes and run the program:



### 1 Click the Save All button on the toolbar.



### 2 Click the Reset Program button on the debugger toolbar.



### 3 Click the Run Project button on the toolbar. Enter values. The program runs. When you enter values and press the Compute Values button, the subtracted value is now correct. However, if you look carefully at remaining results, you'll see that the divided result is also incorrect. Go to Step 6 to find and fix the error.

- 4 Exit the program. Remove the message pane tabs by right-clicking the Application1 tab and choosing Remove “Application1” Tab.

In the next step, you’ll find and fix another runtime error.

## Step 6: Fixing the divideValues() method

---

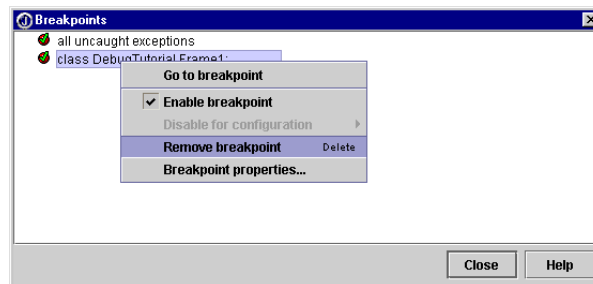
In this step of the tutorial, you will find and fix another of three runtime errors. You will set a breakpoint, step into a method, and learn how to use tool tips and ExpressionInsight to locate errors.

In the previous step, you found and fixed an error with the call to the subtractValues() method. Now, when you run the program again, you may notice that the divided result is also incorrect. For example, if you enter 4 in the Value 1 field and 2 in the Value 2 field, the divided result is 8.0 instead of 2.0.

To find this error, we’ll first set a breakpoint, step into the questionable method, and use ExpressionInsight and tool tips.

- 1 Choose Run | View Breakpoints to remove the breakpoint you set in Step 5. In the Breakpoints dialog box, right-click the following breakpoint:

```
class DebugTutorial.Frame1; line 213; (unverified)
```

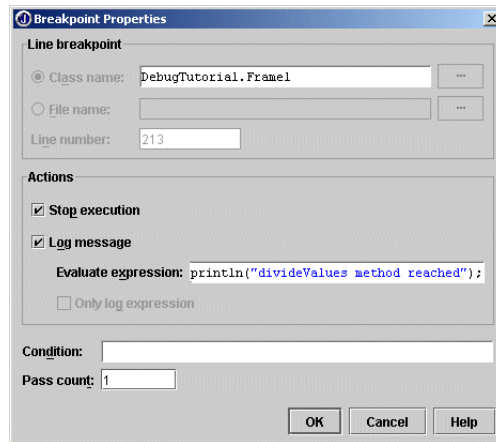


Choose Remove Breakpoint and click the Close button to close the dialog box.

- 2 Use the Find/Replace Text dialog box to locate the call to the divideValues() method.
- 3 Set a breakpoint on this line. Right-click the breakpointed line, and choose Breakpoint Properties to open the Breakpoint Properties dialog box.
- 4 Click the Log Message option. In the Evaluate Expression input field, enter:

```
System.out.println("divideValues method reached")
```

The message will be written to the Console output, input and errors view when the specified breakpoint is reached. If the Stop Execution option is also selected the program will stop. The dialog box will look similar to this:



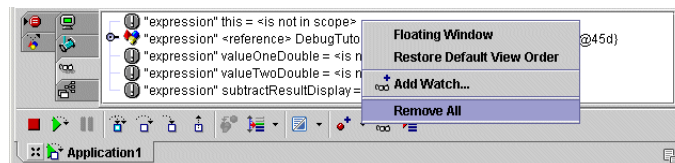
Click OK to close the dialog box.

- 5 Click the Debug button on the main toolbar.
- 6 Enter 4 in the Value 1 input box and 2 in the Value 2 input box when the program's UI is displayed. Press Compute Values. Remember, before you can examine the results, the debugger takes control and is displayed in the message pane.
- 7 Go to the Console output, input and errors view. You'll see the message:

```
Hit breakpoint in class DebugTutorial.Frame1 at line 216
Log Expression: System.out.println("divideValues method reached") = void
```

During the development cycle, you can use this feature instead of adding `println` statements to your code.

- 8 Go to the Data and code watches view. Notice that most of the watches are no longer in scope. Right-click an empty area of the view and choose Remove All.



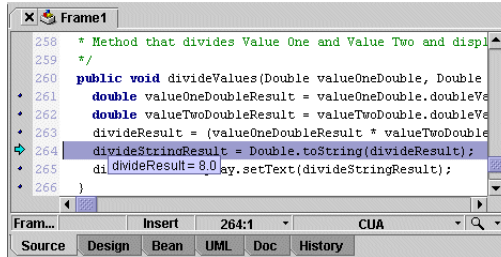
- 9 Click the Step Into button to step into the `divideValues()` method.

## Step 6: Fixing the divideValues() method

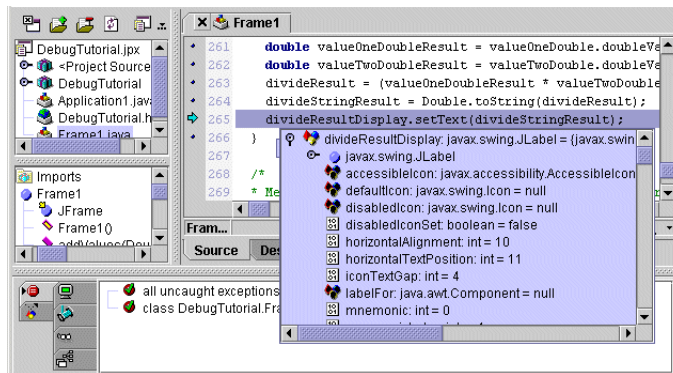
- 10 Click the button three more times, so that you step past the line that reads:

```
divideResult = (valueOneDoubleResult * valueTwoDoubleResult)
```

- 11 Position the mouse over the variable `divideResult` in the editor. A tool tip displaying the value of `divideResult` pops up. Notice that the value is incorrect. Based on what you entered, the result should be 2.0. However, it is 8.0.



You can also press the *Ctrl* key plus right-click the mouse button to display *ExpressionInsight*. This pop-up window shows the expression name, its type, and its value. If the expression is an object, you can descend into the object members, as well as use a right-click menu to set watches, change values, and change base display values. For example, position the cursor over `divideResultDisplay` in the next line of code. Press the *Ctrl* key plus right-click the mouse button. You will see the members of the `JLabel` object. As you scroll down, notice the grayed-out items: these are inherited.



Click in the editor to close the *ExpressionInsight* window. The window will also automatically close if the cursor is repositioned.



- 12** Carefully read this line of source code (the line immediately before the execution point):

```
divideResult = (valueOneDoubleResult * valueTwoDoubleResult)
```

Can you find the error? The `divideResult()` method is multiplying values instead of dividing them.

- 13** To fix the error, change the `*` operator to `/`.

## Saving files and running the program

---

Save your changes and run the program:

- 1** Remove the breakpoint in the editor.
- 2** Click the Save All button on the toolbar.
- 3** Click the Reset Program button on the debugger toolbar.
- 4** Click the Run Project button on the toolbar. Enter values in the Value 1 and Value 2 input fields. The program runs and the divided value is now correct. However, if you look carefully at the remaining results, you may spot the last error. If you enter an odd number in the Value 1 field, the program incorrectly reports that the value is even. If you enter an even value, the program says it is odd.
- 5** Exit the program. Remove the Application1 tab from the message pane.

In the next step, you'll find and fix the last runtime error in this tutorial.

## Step 7: Fixing the oddEven() method

---

In this step of the tutorial, you will find the last of the three runtime errors. You will use the Evaluate/Modify dialog box to evaluate a method call, step into and over a method, set a watch, and change a boolean value on-the-fly to test a theory.

In Step 6, you fixed an error in the `divideValues()` method. Now, when you run the program again, you may notice the statement saying whether the first value is odd or even is incorrect.

For example, if you enter 4 into the Value 1 field, the program reports it is an odd number. However, if you enter 3, the program says that the value is even. In this step, you will find and fix this error.

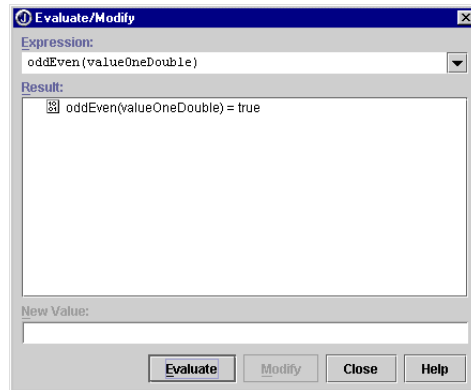
## Step 7: Fixing the `oddEven()` method

To find this error, we'll use the Evaluate/Modify dialog box to evaluate the method that determines if the number is odd or even. Then we'll set a watch on the result returned from the method to see if it's printing to the screen correctly.

- 1 Use the Find/Replace Text dialog box to locate the call to the `oddEven()` method in `Frame1.java`. Notice that a variable name also includes the text `OddEven`. To find the method, you can turn the Case Sensitive option on in the dialog box or search for: `oddEven()`
- 2 Set a breakpoint on this line:  

```
oddEven(valueOneDouble);
```
- 3 Click the Debug button.
- 4 Enter 3 in the Value 1 input box and 4 in the Value 2 input box when the program's UI is displayed. Click the Compute Values button. The focus returns to the debugger.
- 5 Choose Run | Evaluate/Modify to open the Evaluate/Modify dialog box.

**Tip** You can also right-click in the editor and choose Evaluate/Modify. Enter `oddEven(valueOneDouble)` in the Expression input box. Click Evaluate. You'll see that the method returns `true`.



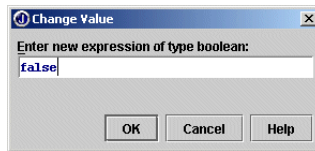
Close the Evaluate/Modify dialog box. Now, we'll step into the method in order to evaluate what the `true` value means.

- 6 Go to the Data watches view. Set a watch on `valueOneIsOdd`.
- 7 Click the Step Into button on the debugger toolbar. When you step into the `oddEven()` method, the value of `valueOneIsOdd` is `true`, because the value was initialized to `true`. (To see the initialization, use the Find Text dialog box to search for `boolean valueOneIsOdd`. Use Run | Show Execution Point to return to the cursor location.)

- 8 Click Step Into three more times to step further into the method. This method determines if the value is odd or even. As you step, the value of `valueOneIsOdd` remains `true`. Is this correct? Does the result of  $(3 \text{ modulus } 2)$  equal zero? It actually does not equal zero, and the value of `valueOneIsOdd` should be set to `false`.

- 9 Right-click `valueOneIsOdd` in the Data watches view and choose Change Value to test this theory. The Change Value dialog box is displayed.

Enter `false` and click OK. The value of `valueOneIsOdd` is set to `false`. You just changed the method's returned value from `true` to `false`.



Click OK to close the dialog box.



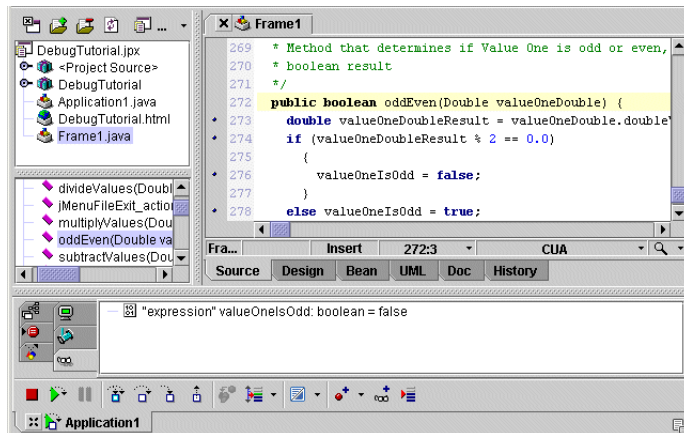
- 10 Click Step Out to step out of the method and return to the calling location, then click Step Into to trace into the `if` statement in the next line of code.

- 11 Examine the contents of the `if` statement. It is actually quite simple:

If `valueOneIsOdd` is `true`, print the message stating that the number is even. However, if the value is `false`, print the message stating that the number is odd.

- 12 Click the Step Into button again. The execution point goes to the `else` statement, the line that states: "If the value of `valueOneIsOdd` is `false`, print the message stating the number is odd."

- 13 Click the `oddEven()` method in the structure pane to go to the location of the method in the editor. (You may have to scroll the structure pane to see the method.)



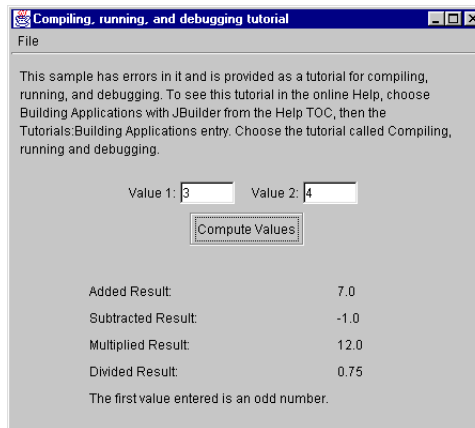
- 14** Examine the modulus operation and its results. Are the `true/false` results assigned correctly? If you look closely, you'll notice that the `true` and `false` assignments are actually mixed up. The code is stating that if the modulus equals zero, the return value is `false` and the number is odd. If the modulus does not equal zero, the return value is `true` and the number is even. These statements should actually be reversed, so that the code will read:

```
if (valueOneDoubleResult % 2 == 0.0)
{
    valueOneIsOdd = true;
}
else valueOneIsOdd = false;
```

- 15** Switch the `true` and `false` values.

Save your changes and run the program:

- 1 Save your files.
- 2 Click the Reset Program button on the debugger toolbar.
- 3 Run the program again.
- 4 Enter 3 in the Value 1 input box and 4 in the Value 2 input box. Click the Compute Values button. The result is correct! The program now correctly informs you that Value 1 is an odd number.



- 5** Click File | Exit to exit the program. Remove the Application1 tab.

In the next step, you will see what happens when a runtime exception is generated.

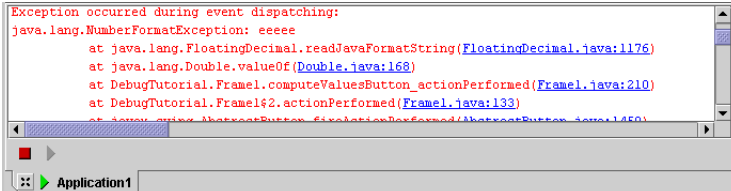
## Step 8: Finding runtime exceptions

In this step of the tutorial, you'll see what happens when a runtime exception is generated. The sample program does not do any error handling. For example, if you enter a character in the Value 1 or Value 2 fields instead of a number, the program will generate a runtime exception stack trace. It won't gracefully tell you that the value was not the expected format or provide information about valid values.

To see what a runtime exception stack trace looks like,

- 1 Run the program.
- 2 Enter `eeee` in the Value 1 input field. Enter `3` in the Value 2 input field. Press Compute Values.
- 3 Minimize the program to view the message pane.

The Application1 tab now displays a `NumberFormatException` stack trace. This is a trace of how your program arrived at this exception.



```
Exception occurred during event dispatching:
java.lang.NumberFormatException: eeeee
    at java.lang.Float.parseFloat(Float.java:1176)
    at java.lang.Double.valueOf(Double.java:168)
    at DebugTutorial.Frame1.computeValuesButton_actionPerformed(Frame1.java:210)
    at DebugTutorial.Frame1$2.actionPerformed(Frame1.java:133)
    at javax.swing.JButton.actionPerformed(JButton.java:146)
```

- 4 Click the first underlined class name in the stack trace to see where the exception is thrown. In this case, click `Float.parseFloat`.

JBuilder opens the source code for `java.lang.Float.parseFloat` and highlights the line of code where the exception is thrown. You can click other classes in the stack trace to trace through the steps that brought the program to this exception.

To handle this exception is beyond the scope of this tutorial. To run the program again without the exception, just close the program and run it again entering numeric values.

Congratulations, you have finished this tutorial. You found and fixed syntax errors, compiler errors, and runtime errors using JBuilder's integrated debugger. You also saw an example of a runtime exception stack trace.

For more information on compiling, running, and debugging, read the following chapters:

- [Chapter 6, "Building Java programs"](#)
- [Chapter 5, "Compiling Java programs"](#)
- [Chapter 7, "Running Java programs"](#)
- [Chapter 8, "Debugging Java programs"](#)



# Chapter 18

## Tutorial: Building with Ant files

This tutorial uses features  
in JBuilder Enterprise

This tutorial explains how to work with Ant build files to build your projects. Ant is a Java-based build tool that builds projects as specified by one or more XML build file. The build files define build targets and build tasks. For example, a build file might contain separate targets for building a project and generating Javadoc. You can execute individual targets or the default target for the project using the Ant build file.

JBuilder automatically recognizes Ant build files named `build.xml` and displays these nodes with an Ant icon instead of the usual XML icon. You can also use the Ant wizard to import Ant files of any name. The targets in the `build.xml` file display as child nodes.

In this tutorial, you'll complete the following tasks:

- Create a project and application.
- Create an Ant build file.
- Execute Ant build targets.
- Execute the default Ant target.
- Introduce an error in a Java source file and examine the Ant error messages.
- Add a target to the Project menu.
- Modify Ant properties.
- Add custom Ant libraries.

### See also

- “Building with external Ant files” on page 6-7
- The Jakarta project at Apache: <http://jakarta.apache.org/ant>
- Ant documentation in the JBuilder `extras/ant/docs/` directory

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see ["Documentation conventions" on page 1-4](#).

## Step 1: Creating a project and application

---

In this step, you'll use JBuilder wizards to create a project and application.

- 1 Choose File | New Project to open the Project wizard.
- 2 Enter `AntProject` in the Name field and click Finish to close the wizard and create the project.
- 3 Choose File | New to open the object gallery and double-click the Application icon on the General page to open the Application wizard.
- 4 Accept the defaults and click Finish to close the wizard.

## Step 2: Creating the Ant build file

---

Now that you have a project to build, you'll create an Ant build file and use it to build the project. JBuilder automatically recognizes files named `build.xml` as Ant build files and displays Ant icons for those nodes in the project pane.

First, create the Ant build file.

- 1 Choose File | New File.
- 2 Enter `build` in the Name field, choose XML as the file extension from the Type drop-down list, and click OK. The new file is open in the editor.
- 3 Click OK to save the new file to your project.
- 4 Enter the following text or copy and paste it into the editor:

```
<?xml version="1.0"?>
<!DOCTYPE project>
<project name="AntProject" default="dist" basedir=".">
  <property name="src" value="src"/>
  <property name="build" value="build"/>
  <property name="dist" value="dist"/>

  <target name="init">
    <tstamp/>
    <mkdir dir="${build}"/>
  </target>
```



```

<target name="compile" depends="init">
<javac srcdir="${src}" destdir="${build}"/>
</target>

<target name="dist" depends="compile">
<mkdir dir="${dist}/lib"/>
<jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
</target>

<target name="clean">
<delete dir="${build}"/>
<delete dir="${dist}"/>
</target>

</project>

```

- 5 Save the project.
- 6 Choose Project | Add Files/Packages, browse to the `AntProject` directory on the Explorer tab, select `build.xml`, and click OK to add it to your project.
- 7 Examine the `build.xml` file to understand what it does:
  - project: includes a project name, the default target to run if none of the other individual targets are run, and the location of the base directory.
  - properties: Ant targets and tasks are typically “property-aware.” Properties are also used to pass parameters to tasks without overriding the existing properties in the build file.
  - init target: creates a `build` directory for the compiled classes.
  - compile target: initiates the init target first, then compiles the Java source files and puts the generated `.class` files in the `build` directory.
  - dist target: initiates the compile target first, then makes a `dist/lib/` directory, and creates a JAR file in that directory.
  - clean target: deletes the `build` and `dist` directories.

## Step 3: Executing individual targets

---

Next, you’ll run two targets in the `build.xml` file. First, you’ll run the `init` target to create the `build` directory for the compiled `.class` files. Then, you’ll run the `clean` target to remove the output and the `build` directory.



- 1 Click the Refresh button on the project pane toolbar and expand the Ant node in the project pane to expose the child nodes, which are the Ant targets.

## Step 4: Executing the default target

- 2 Right-click the Ant init target in the project pane and choose Make. This target creates the build directory, where the compiled .class files will go.
- 3 Examine the messages output to an Ant node in the message pane. The init target successfully created the build directory.

```
StdOut
Buildfile: build.xml
init:
[mkdir] Created dir: C:\Documents and Settings\ktaylor\jbproject\
      AntProject\build
BUILD SUCCESSFUL
Total time: 2 seconds
```

- 4 Right-click the Ant compile target in the project pane and choose Make. This target compiles the .java source files, generates the .class files, and puts them in the build directory created by the init target.
- 5 Right-click the Ant clean target and choose Make to remove all the build output, including the build directory.

## Step 4: Executing the default target

---

If you choose the Ant node and do a Make on it, JBuilder runs the default Ant target, in this case, the dist target. The default target is specified in the <project> element. The dist target creates the dist/lib/ and generates a JAR file in that directory. Notice that the dist target depends on compile, which depends on init. So when the dist target is executed, it executes the compile target, which in turn executes the init target. Due to these dependencies, the execution order of targets is: init, compile, dist.

Next, you'll execute the default target in the build file.

- 1 Right-click the Ant build.xml node and choose Make to execute the default target, dist.
- 2 Look in the message pane to see the results of the build:

```
StdOut
Buildfile: build.xml
init:
[mkdir] Created dir: C:\Documents and Settings\ktaylor\
      jbproject\AntProject\build
compile:
[javac] Compiling 2 source files to C:\Documents and Settings\ktaylor\
      jbproject\AntProject\build
dist:
[mkdir] Created dir: C:\Documents and Settings\ktaylor\
      jbproject\AntProject\dist\lib
[jar] Building jar: C:\Documents and Settings\ktaylor\jbproject\
      AntProject\dist\lib\MyProject-20020826.jar
BUILD SUCCESSFUL
Total time: 4 seconds
```

Because you previously removed the classes and their directory with the `clean` target, the `init` target recreates the `build` directory. The `compile` target once again compiles the `.class` files. Lastly, the `dist` target creates the `dist/lib/` directory and generates a JAR file in that directory.

## Step 5: Handling errors with Ant

---

In this step, you'll introduce an error in a Java source file, make the Ant build file, and use the error messages to navigate to the error in the source file.

- 1 Open `Application1.java` in the editor.
- 2 Find the `main()` method and add comment tags as shown:

```
//public static void main(String[] args) {
```

- 3 Right-click `build.xml` and choose `Make`.
- 4 Examine the message pane and notice that it displays a `StdErr` node that displays error messages. In this example, adding comment tags before the `main()` method produces two errors:

```
StdErr
"Application1.java":      [javac] C:\Documents and Settings\ktaylor\jbproject\
                          AntProject\src\antproject\Application1.java:43: error #203: illegal
                          start of type at line 43
                          [javac]   try {
                          [javac]   ^
"Application1.java":      [javac] C:\Documents and Settings\ktaylor\jbproject\
                          AntProject\src\antproject\Application1.java:51: error #202: 'class' or
                          'interface' expected at line 51
                          [javac] }
                          [javac] ^
```

- 5 Choose an error message in the message pane to highlight it in the editor. Double-click an error message to move the cursor to the line of code in the editor.

**Tip** If you are using the Borland Java compiler (**bmj**) to build your Ant file, compiler errors are listed by number. For a complete list of compiler errors by number, see “Compiler error messages” in online help. The Use Borland Java Compiler option is on by default. See [“Setting Ant properties” on page 6-12](#).

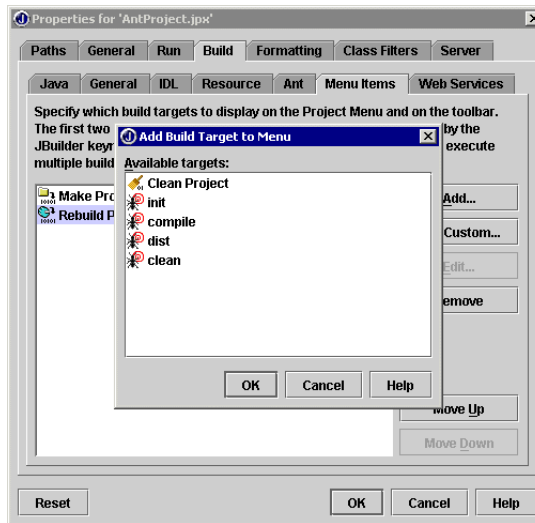
- 6 Remove the comment tags from the `main()` method before going to the next step.

## Step 6: Adding a target to the Project menu

---

In this step, you'll add the Ant clean target to the Project menu and reorder the build targets on the Project menu. For more information on configuring this menu, see [“Configuring the Project menu” on page 6-18](#).

- 1 Choose Project | Project Properties to open the Project Properties dialog box.
- 2 Choose the Build tab, then the Menu Items tab.
- 3 Click the Add button to open the Add Build Target To Menu dialog box.



- 4 Choose the Ant target, clean (build.xml), **not** the JBuilder target, Clean.
- 5 Click OK to add clean as a build target on the Project menu.
- 6 Choose the Move Up button to move the Ant clean target up in the list below Make. Now, the clean target will be the second menu item on the Project menu.

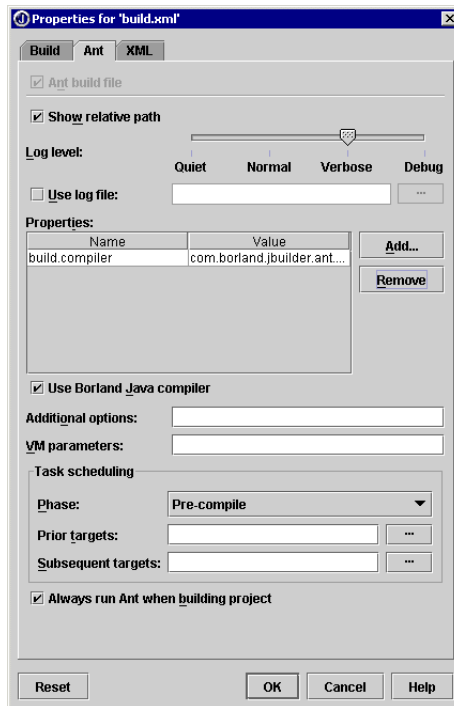
**Tip** The first two menu items on the Project menu have configurable key bindings, which you can modify in the Keymap Editor (Tools | Editor Options | Editor | Customize).

- 7 Choose OK to close the Project Properties dialog box.
- 8 Choose Project | clean to clean the project. Notice in the message pane that the clean target was executed.

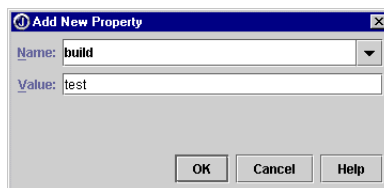
## Step 7: Setting Ant properties

There may be cases where you want to change Ant properties in the build file without overwriting it. You can do this by passing parameters in the Ant properties dialog box.

- 1 Right-click the Ant `build.xml` node and choose Properties.
- 2 Choose the Ant tab and change the Log Level option to Verbose, which provides more information in the message pane.



- 3 Click the Add button to the right of the Properties list.
- 4 Choose `build` from the Name drop-down list and enter `test` in the Value field.



**5** Click OK twice to close both dialog boxes.

Now, when you execute the Ant compile target, a test directory is created and the class files are created in the test directory instead of the build directory.

**6** Right-click the Ant compile target and choose Make.

**7** Examine the output in the message pane. There is more information, because the verbose option is used. The results tell you that the test directory was created when the init target executed, instead of the build directory. You should see something similar to this in the message pane:

```
StdOut
Apache Ant version 1.5 compiled on July 9 2002
Buildfile: build.xml
Detected Java version: 1.4 in: C:\jbuilder\jdk1.4\jre
Detected OS: Windows 2000
parsing buildfile build.xml with URI = file:C:/Documents and Settings/ktaylor\
/jbproject/AntProject/build.xml
Project base dir set to: C:\Documents and
Settings\ktaylor\jbproject\AntProject
Override ignored for property build
Build sequence for target 'compile' is [init, compile]
Complete build sequence is [init, compile, clean, dist]
init:
[mkdir] Created dir: C:\Documents and Settings\ktaylor\
jbproject\AntProject\test
compile:
[javac] antproject\Application1.java added as C:\Documents and
Settings\ktaylor\
jbproject\AntProject\test\antproject\Application1.class doesn't exist.
[javac] antproject\Framel.java added as C:\Documents and Settings\ktaylor\
jbproject\AntProject\test\antproject\Framel.class doesn't exist.
[javac] Compiling 2 source files to C:\Documents and Settings\ktaylor\
jbproject\AntProject\test
BUILD SUCCESSFUL
Total time: 4 seconds
```

Next, you'll set an option to always build the project with Ant when you use the Project Make or Project Rebuild command. First, you'll clean the project.

**1** Choose Project | clean. As you can see from the message pane, all of the Ant output is deleted, including the class files and the test directory.

**2** Right-click AntProject.jpx in the project pane and choose Clean. This removes the classes in the classes directory that the JBuilder build system generated. If you look in your operating system's file manager, you'll see that the classes and the classes directory generated by JBuilder have been deleted.

**3** Right-click the Ant build.xml node and choose Properties.

- 4 Click the Ant tab and check the Always Run Ant When Building Project option on the Ant page and click OK to close the dialog box. Now when you choose Project | Make Project, Ant runs as part of the JBuilder build process.
- 5 Choose Project | Make Project to build the project.

Ant runs the default Ant target and JBuilder builds with Make. The Ant messages displayed in the message pane tell you that new directories were created, classes compiled, and a JAR created. Look in your operating system's file manager to see that JBuilder also generated the class files in the `classes` directory when Make was executed.

## Step 8: Adding custom Ant tasks to your project

---

There may be cases in which you have custom libraries that contain custom Ant build tasks. For example, you might have build tasks in your Ant build file that need to execute tools such as ANTLR Translator generator, Java mail, or JUnit testing. You can create a custom library that includes the paths to these tools and add it to your project. You can also use a different version of Ant by adding a library with the Ant JARs. If you don't specify any Ant JARs, JBuilder uses the Ant delivered in the JBuilder `lib` directory.

You can add these libraries to your project on the Build page of Project Properties as follows:

- 1 Choose Project | Project Properties.
- 2 Choose the Build tab and then the Ant tab.
- 3 Click the Add button to open the Select A Library dialog box.
- 4 Select an existing library in the list or click the New button to open the New Library wizard and create a library. Click OK to close the Select A Library dialog box and add the library to the project. For more information on creating libraries, see [“Adding projects as required libraries” on page 3-4](#).
- 5 Select a library in the list and choose the Move Up or Move Down button if you want to change its order in the list. Libraries are searched in the order listed.
- 6 Click OK to close the Project Properties dialog box.

Now that you've completed the tutorial, see [“Building with external Ant files” on page 6-7](#) to learn more about running Ant build files in JBuilder.





# Chapter 19

## Tutorial: Remote debugging

This tutorial is a feature of  
JBuilder Enterprise

This step-by-step tutorial shows you how to

- Use remote debugging features to attach to a program already running on a remote computer.
- Debug using cross-process stepping.
- Use preset configurations to debug both a client and server process.

The tutorial uses the sample project that is provided in the `<jbuilder>\samples\RMI` folder. The sample is an RMI application, created in JBuilder. Before running this tutorial, make sure that you have installed the `samples` folder.

This tutorial assumes the following:

- You are using a Windows computer.
- You are familiar with compiling, running, and debugging. If not, work through the tutorial in [Chapter 17, “Tutorial: Compiling, running, and debugging.”](#) You can also read the following chapters:
  - [Chapter 6, “Building Java programs”](#)
  - [Chapter 5, “Compiling Java programs”](#)
  - [Chapter 7, “Running Java programs”](#)
  - [Chapter 8, “Debugging Java programs”](#)
- You are familiar with client/server processes in JBuilder.
- You have read [Chapter 9, “Remote debugging.”](#)
- You are comfortable with DOS windows and with running commands from the command line.

## Step 1: Opening the sample project

To run this tutorial, you need

- Two computers running on a network. JBuilder must be installed on one; JDK 1.3 or higher must be installed on the other. In this tutorial, the computer with JBuilder will be called the “client” computer. The computer with just the JDK will be called the “remote” computer. This computer will run the server.
- The host name or IP address of the remote computer. This ID is usually set up by the network administrator.
- A way to transfer files from the client computer to the remote computer.

**Note** To run the sample without debugging it, follow the instructions in the project’s HTML file, `SimpleRMI.html`.

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 1-4](#).

## Step 1: Opening the sample project

---

This tutorial uses the sample project that is provided in the `samples\RMI` folder of your JBuilder installation. Before running this tutorial, make sure that you have installed the `samples` folder.

In this step, you will open the project file. To open the sample project,

- 1 Choose **File | Open Project**. The Open Project dialog box is displayed.
- 2 Navigate to the `<jbuilder>\samples\RMI` folder.
- 3 Double-click `SimpleRMI.jpx`. The project is opened in the project pane. The files in the project are listed in the project pane. This project consists of six files:
  - `SimpleRMI.html` - The HTML file that provides a descriptive overview of the project. This file provides instructions on creating an RMI application in JBuilder and running it.

- `SimpleRMI.policy` - The security policy file. This file specifies the rights of the RMI server to listen for and accept client requests over a network.
- `SimpleRMIClient.java` - The client class that connects to the server object.
- `SimpleRMIIImpl.java` - The class that implements the RMI server interface.
- `SimpleRMIInterface.java` - The RMI interface.
- `SimpleRMIServer.java` - The server class that creates an instance of the `Impl` class.

In Step 2, you will set client and server runtime and debug configurations.

## Step 2: Setting runtime and debugging configurations

---

In this step, you will set runtime and debugging configurations for the client and server. For more information on runtime and debugging configurations, see [“Setting runtime configurations” on page 7-6](#) and [“Setting debug configuration options” on page 8-68](#).

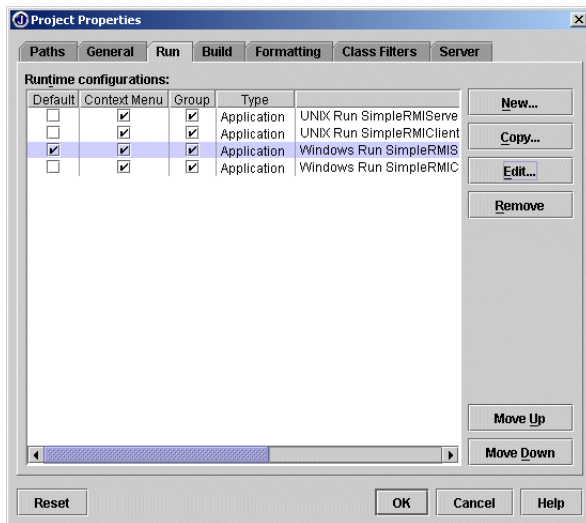
To set the configurations for this tutorial, use the dialog box pages listed in the following table.

**Table 19.1** Dialog box pages for setting client and server runtime and debugging configurations

Runtime Configuration Properties dialog box page	Applies To	Description
Run page	Server (runs on the remote computer)	Configures the run parameters for the RMI server.
Debug page	Server (runs on the remote computer)	Configures how the server on the client computer attaches to the remote server process.
Run page	Client (runs on the computer with JBuilder)	Configures the run parameters for the RMI client.

To set runtime configurations for the server,

- 1 Choose Run | Configurations. The Run page of the Project Properties dialog box is displayed.



- 2 Choose the configuration called Windows Run SimpleRMIServer.
- 3 Press Edit to display the Run page of the Runtime Configuration Properties dialog box.
- 4 Make sure the VM Parameters codebase argument points to the location of the server class files. In a typical Windows installation, this will be the `classes` folder in the `<jbuilder>\samples\RMI` folder:

```
-Djava.rmi.server.codebase=file:C:\<jbuilder>\samples\RMI\classes\
```

**Note** The last backslash in the argument, after the `classes` entry, is required.

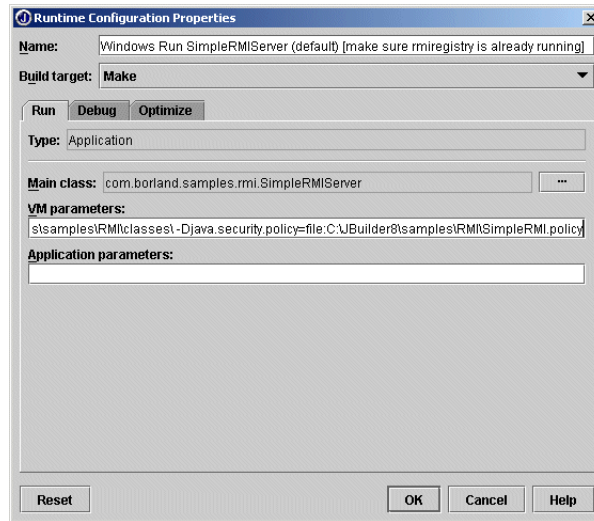
- 5 Make sure the security policy argument in the VM Parameters field points to the location of the security policy file. The policy file specifies the rights of the RMI server to listen for and accept RMI client requests over a network. In a typical Windows installation, this will be the `<jbuilder>\samples\RMI` folder.

```
-Djava.security.policy=file:C:\<jbuilder>\samples\RMI\SimpleRMI.policy
```

- 6 Make sure the main class is set to:

```
com.borland.samples.rmi.SimpleRMIServer
```

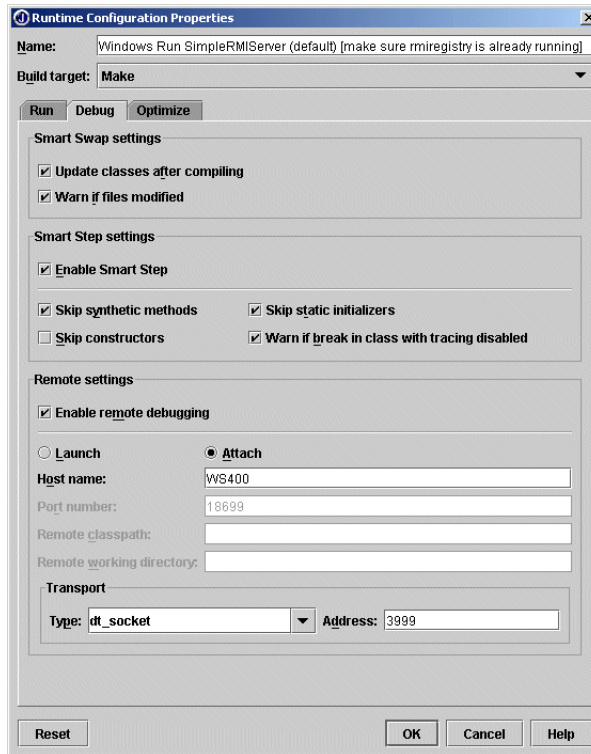
- 7 When you're finished, the Run page for the server should look similar to this:



To set the remote debugging configuration for the server,

- 1 Click the Debug tab.
- 2 Click the Enable Remote Debugging option and then the Attach option.
- 3 Enter the name of the computer where the server will be running in the Host Name field.
- 4 Leave the Transport Type as `dt_socket`.
- 5 Enter the address of the remote computer in the Address field. You will be using this number again when you run the server on the remote computer (in ["Step 5: Starting the RMI Registry and server on the remote computer" on page 19-10](#)). For the purposes of this tutorial, leave this set to `3999`.

- 6 When you're finished, the Debug page for the server should look similar to this:



- 7 Click OK to close the Runtime Configuration Properties dialog box for the server.

Next, you'll set runtime configurations for the client.

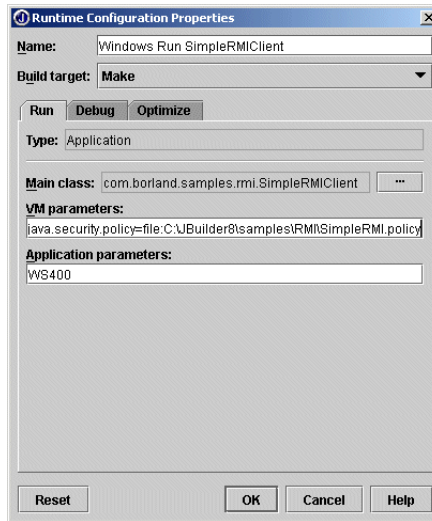
- 1 On the Run page of the Project Properties dialog box, choose the configuration called `Windows Run SimpleRMIClient`.
- 2 Press Edit to display the Run page of the Runtime Configuration Properties dialog box.
- 3 Make sure the argument in the VM Parameters field points to the location of the security policy file. In a typical Windows installation, this will be the `<jbuilder>\samples\RMI` folder.  

```
-Djava.security.policy=file:C:\<jbuilder>\samples\RMI\SimpleRMI.policy
```
- 4 Make sure the main class is set to:  

```
com.borland.samples.rmi.SimpleRMIClient
```
- 5 In the Application Parameters field, enter the name of the remote computer. This is the name you entered into the Host Name field of the

Debug page of the Runtime Configuration Properties dialog box for the server (see the previous section).

- 6 When you're finished, the Run page for the client should look similar to this:



- 7 Click OK to close the Runtime Configuration Properties dialog box.

- 8 Click OK again to close the Project Properties dialog box.

In the next step, you will set the breakpoints for the client and the server.

## Step 3: Setting breakpoints

---

In this step, you will set a line breakpoint in the client process and a cross-process breakpoint in the server process. The line breakpoint will cause the client to pause when the cross-process breakpoint is about to be called. The cross-process breakpoint will pause the server. This technique allows you to step into a server process from a client process.

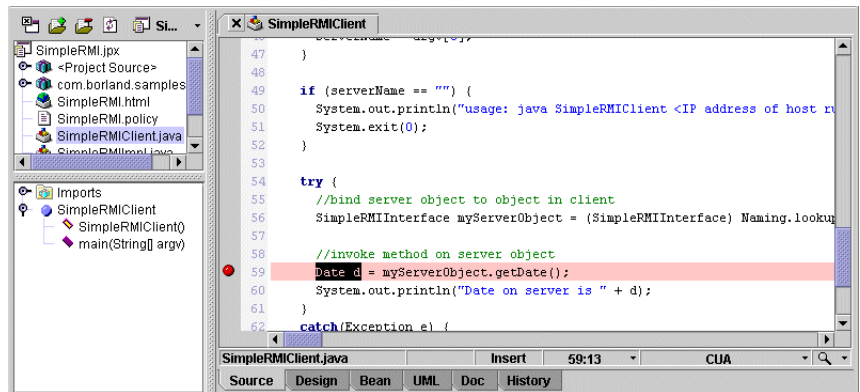
To set a line breakpoint in the client process,

- 1 Double-click `SimpleRMIClient.java` in the project pane. It is opened in the editor.
- 2 Use the Search | Find to find the string `Date d`. The cursor will be placed on the following line:

```
Date d = myServerObject.getDate();
```

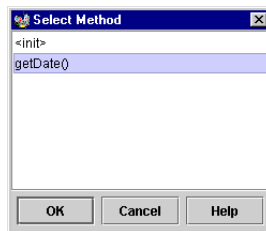
### Step 3: Setting breakpoints

- 3 Click the gutter, the gray area to the left of the line of code, to set a breakpoint on the line.



To set a cross-process breakpoint in the server process,

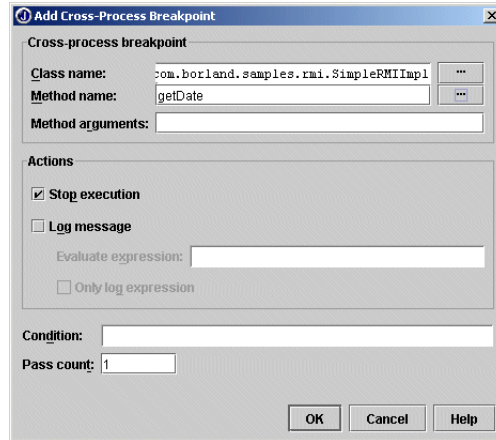
- 1 Choose Run | Add Breakpoint | Add Cross Process Breakpoint. The Add Cross-Process Breakpoint dialog box is displayed.
- 2 Choose the ellipsis button to the right of the Class Name field.
- 3 In the Search For field of the Select Classes dialog box, enter `SimpleRMIIImpl`.
- 4 Click OK to close the dialog box when the class is selected.
- 5 Choose the ellipsis button to the right of the Method Name field.
- 6 Choose `getDate()` in the Select Method dialog box.



- 7 Click OK to close the dialog box.
- 8 Leave the Actions option in the Add Cross-Process Breakpoint dialog box set to Stop Execution.



9 The Add Cross-Process Breakpoint dialog box should look like this:



10 Click OK to close the dialog box.

In the next step, you will compile the server and copy the server class files to the remote computer.

## Step 4: Compiling the server and copying server class files to the remote computer

---

This step tells you how to compile the server and copy the server class files to the remote computer.

To compile the server files in JBuilder, choose Project | Make Project "SimpleRMI.jpx". The status bar shows when the project has been built.

Notice that an expand/collapse icon is displayed by `SimpleRMIImpl.java` in the project pane. The RMI compiler created the stub class, `SimpleRMIImpl_Stub.java`. Do not edit this file as it is auto-generated.

Go to a DOS window and look in the `<jbuilder>/samples/RMI` folder. The folder should now contain a `classes` folder. The `classes` folder contains a hierarchy of folders that follow the package structure. The server `.class` files are stored in the `classes/com/borland/samples/rmi` folder. The `classes` folder also includes the `dependency cache` and `Generated Source` folders.

You need to copy the server class files to the remote computer. For the purposes of this tutorial, you can copy the entire `RMI` folder to the remote computer, to a new folder called `RMI`. To do this, you can either:

- Copy files to a network, then copy them to the remote computer.
- Copy files to diskette and copy them to the remote computer.
- FTP files to the remote computer.

**Important** From this point forward, if you update source files on your client computer (the one running JBuilder), you must re-copy the .class files to the remote computer. If you fail to do so, your source files and compiled files will not match, causing invalid errors.

In the next step, you'll start the RMI Registry and the server on the remote computer.

## Step 5: Starting the RMI Registry and server on the remote computer

---

This step tells you how to start the RMI registry on the remote computer and start the server in debug mode on the remote computer. You need to be aware of the RMI settings as well as the debug settings in the Java command line that starts the server.

To start the RMI registry on the remote computer,

- 1 Open a 4DOS or 4NT window.
- 2 Change to the <jdk>\bin folder.
- 3 Start the RMI Registry by entering the following command:

```
start rmiregistry
```

The RMI Registry starts in a separate process. If the registry does not start, you may be out of available memory. Exit other applications that may be running, then close the DOS window and try again.

To start the server on the remote computer,

- 1 Start a Command window. The Java command line is more than 256 characters; you will not be able to run it in a standard 4DOS or 4NT window. Start the Command window from the Start menu: click Run and enter `command`. For NT computers, enter `cmd`.
- 2 Make sure the <jdk>\bin folder is in your path.
- 3 Go to the root of the folder that contains the RMI sample.
- 4 Enter the following command at the prompt. This command will start the server in debug mode and suspend its execution. You may want to place the command in a batch file or shell script. If you do, make sure the command contains no line breaks.

```
java -Xdebug -Xnoagent -Djava.compiler=NONE  
-Djava.rmi.server.codebase=file:\rmi\classes\  
-Djava.security.policy=file:\rmi\SimpleRMI.policy  
-Xrunjdwp:transport=dt_socket,server=y,address=3999,suspend=y -classpath  
d:\rmi\classes\ com.borland.samples.rmi.SimpleRMIServer
```

The command line you enter to run the server takes both RMI and debugger arguments. A description of each parameter follows.

**Table 19.2** Command line RMI and debugger arguments

Parameter	Description
java	The command to run the Java VM.
-Xdebug	Runs the VM in debug mode.
-Xnoagent	Does not use debug agent.
-Djava.compiler=NONE	Does not use any JITs.
-Djava.rmi.server.codebase=file:\rmi\classes\	Identifies the location of the server's class files.
-Djava.security.policy=file:\rmi\SimpleRMI.policy	Identifies the location of the java security policy file.
-Xrunjdwp:transport=dt_socket,server=y, address=3999,suspend=y	Debugger options, where: <ul style="list-style-type: none"> <li>• <code>transport</code>: The transport method. Needs to match what is set on the Runtime Properties Debug page for the server - see <a href="#">“Step 2: Setting runtime and debugging configurations” on page 19-3</a>.</li> <li>• <code>server</code>: Runs the VM in server mode.</li> <li>• <code>address</code>: The port number through which the debugger communicates with the remote computer. Needs to match what is set on the Runtime Properties Debug page for the server - see <a href="#">“Step 2: Setting runtime and debugging configurations” on page 19-3</a>.</li> <li>• <code>suspend</code>: Indicates whether the program is suspended immediately when it is started.</li> </ul>
-classpath d:\rmi\classes\	The class path.
com.borland.samples.rmi.SimpleRMIServer	The runnable server file (includes the package name).

In the next step, you'll use the debugger to attach to this running server and step into the server's `getDate()` method where the cross-process breakpoint was set.

## Step 6: Starting the server process and the client in debug mode and stepping into the cross-process breakpoint

This step tells you how to start both the server process and the client in debug mode in JBuilder, and then step into the cross-process breakpoint. Once you've started stepping, JBuilder allows you to step between the client and server. You will:

- Start the server process on the client computer in debug mode.
- Start the client on the client computer in debug mode.
- Step into the cross-process breakpoint on the server running on the remote computer.

## Step 6: Starting the server process and the client in debug mode

To start the server process in debug mode on the client computer (the computer running JBuilder),



- 1 Click the down arrow to the right of the Debug Program button on the main toolbar.

- 2 Choose the Windows Run SimpleRMIServer configuration.

### Note

You do not need to start the RMI Registry on the client computer. It's already running on the remote computer.

- 3 The debugger starts and pauses execution.



- 4 Click the Resume Program button on the debugger toolbar.

The message `SimpleRMIServer ready` is displayed on the remote computer. The name of the computer and the address are displayed on the debugger tab at the bottom of the JBuilder AppBrowser window.

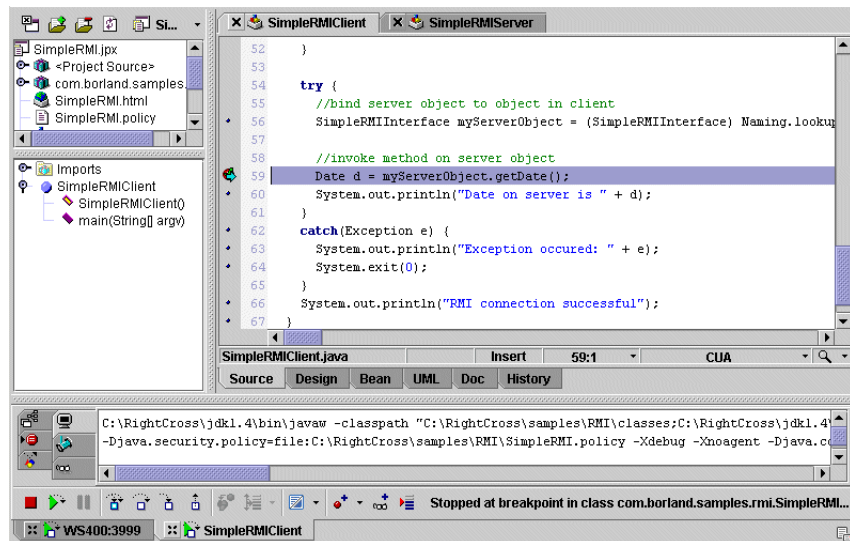
To start the client in debug mode on the client computer (the computer running JBuilder),



- 1 Right-click the down arrow to the right of the Debug Program button on the main toolbar.

- 2 Choose the Windows Run SimpleRMIClient configuration.

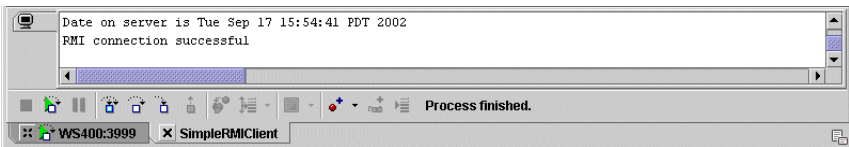
- 3 The debugger starts, and stops execution at the call to the server's `getDate()` method. (You set a breakpoint on this line in Step 2.)



Step 6: Starting the server process and the client in debug mode

To step into the cross-process breakpoint,

- 1 Click the debugger tab for the SimpleRMIClient process.
- 2 Click the Step Into icon on the client's debugger toolbar to step into the server-side breakpointed method. If you use Step Over, the debugger will not stop.
- 3 Click the Step Into button two more times. The message SimpleRMIIImpl getDate() is displayed on the remote computer.
- 4 Continue to click Step Into on the SimpleRMIClient tab until the client runs to completion. The SimpleRMIClient process in the debugger will look like this:




The output from the server running on the remote computer will look like this:

```
SimpleRMIIImpl ready
SimpleRMIIImpl.getDate()
```

- 5 To exit the server on the remote computer, press *Ctrl + C* from the Command window. To close the RMIRegistry, click the close button on the RMIRegistry window.

While starting the server or client in debug mode in JBuilder, you may see one of the following error messages:

Table 19.1 RMI client/server error messages

Error message	Description
connection refused	The RMI Registry on the remote computer might not yet be running. Stop all processes and run the RMI Registry on the remote computer by entering <code>start rmiregistry</code> from the command line. (The <code>&lt;jdk&gt;\bin</code> folder must be in your path.) Restart the remote server and begin the debug process again.
Java exception: java.rmi.NotBoundException SimpleRMIIImpl	You haven't yet started the server debug process. Click the Resume Program button  on the server's debugger toolbar. Start the client again in debug mode.

Congratulations! You have completed the tutorial. Using preset runtime configurations, you ran a RMI server on a remote computer. You then debugged the program using JBuilder's remote debugging features.



# Chapter 20

## Tutorial: Visualizing code with the UML browser

This tutorial uses features in JBuilder Enterprise.

This step-by-step tutorial shows you how to use JBuilder's UML features to navigate and analyze your code. UML is helpful in examining code, analyzing application development, and communicating software design. JBuilder uses UML diagrams for visualizing code and browsing classes and packages. UML diagrams can help you quickly grasp the structure of unknown code, recognize areas of over-complexity, and increase your productivity by resolving problems more rapidly.

For more information on UML features in JBuilder, see [Chapter 11, "Visualizing code with UML."](#) For definitions of UML terms, see ["Java and UML terms" on page 11-2.](#)

In this tutorial, you'll accomplish such tasks as:

- Viewing a UML package diagram
- Viewing a UML class diagram
- Adding library references
- Filtering UML diagrams

The tutorial uses the sample project that is provided in the `samples/DataExpress/ProviderResolver` folder of your JBuilder installation. Before running this tutorial, make sure that you have installed the `samples` folder. For users with read-only access to JBuilder samples, copy the `samples` directory into a directory with read/write permissions.

This sample demonstrates how to build a custom DataExpress Provider and Resolver. It uses a simple application which displays the data provided by `ProviderBean` to the `TableDataSet` in a `JdbTable`. It also includes a `JdbNavToolBar` whose Save button can be pressed to save changes back to

## Step 1: Compiling the sample

the text file, which contains sample data, via `ResolverBean`. For more information on the sample, read the project notes file in the sample: `ProviderResolver.html`. The sample includes the following files:

- `ProviderBean.java`: provides data from a simple, un delimited text file into a `TableDataSet`.
- `ResolverBean.java`: replaces the data in the original text file.
- `TestApp.java`: a simple application which displays the data provided by `ProviderBean` to the `TableDataSet` in a `JdbTable`. It also includes a `JdbNavToolBar` whose Save button can be pressed to save changes back to the text file via `ResolverBean`.
- `TestFrame.java`: the application UI.
- `data.txt`: the text file with some sample data in it.
- `DataLayout.java`: an interface that describes the structure of `data.txt`.

### See also

- [“Java and UML terms” on page 11-2](#)
- [“JBuilder UML diagrams defined” on page 11-7](#)

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 1-4](#).

## Step 1: Compiling the sample

---

In this step, you’ll compile the project. It’s always best to compile before you choose the UML tab, so the UML diagram is up-to-date and accurate. When you choose the UML tab, JBuilder loads the class files to determine their relationships, which the UML browser then uses to obtain the package and class information for the UML diagrams.

Begin by opening the sample:

- 1 Choose **File | Open Project** and browse to the `ProviderResolver` sample:  
`<jbuilder>/samples/DataExpress/ProviderResolver/ProviderResolver.jpx`
- 2 Choose **Project | Make Project** to compile the project.

### Important

Before viewing the UML diagram for a project or file, you must always compile the project to see the complete UML diagram. JBuilder then loads the compiled classes to build the diagrams. Choose **Project | Make Project** before choosing the UML tab.



## Step 2: Viewing a UML package diagram

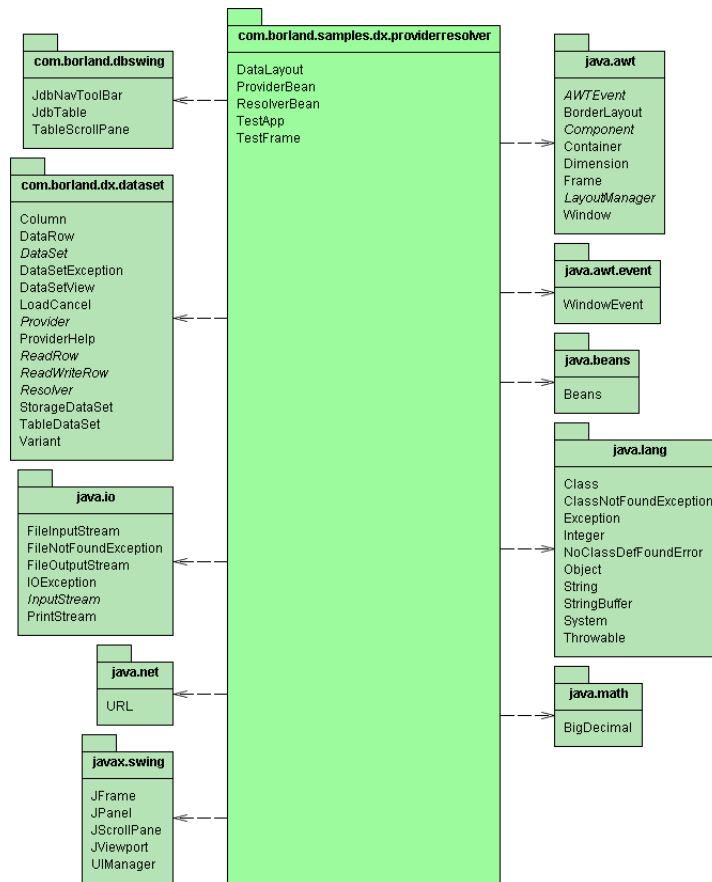
Now that the project is compiled, JBuilder can create the UML diagrams from the generated class files. Begin by looking at the UML package diagram.

- 1 Double-click the `com.borland.samples.dx.providerresolver` package node in the project pane to open it in the content pane. It opens by default with the Package tab active, which displays the package summary.

### Note

If the package node is not available, set the Enable Source Package Discovery And Compilation option on the General page of the Project Properties dialog box (Project | Project Properties).

- 2 Choose the UML tab to view the UML package diagram. When you choose the UML tab, JBuilder loads the classes to determine their relationships and builds the UML diagram.

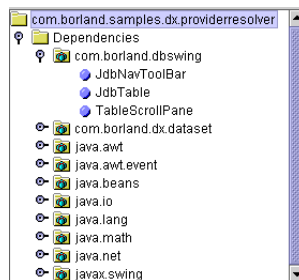


The UML package diagram represents a package as a rectangle with a tab and the full package name at the top. The current package, `com.borland.samples.dx.providerresolver`, is in the center with its dependencies on other packages to either side. Dependencies in the UML diagram are represented by a dashed line pointing from the central package to the package it's dependent on. This central package, which has a bright green background, lists all of the classes within it. The outer packages, which have a darker green background, list only the classes that `com.borland.samples.dx.providerresolver` is dependent on.

- 3 Examine the structure pane to the lower left of the UML diagram. There are two folders that can appear with package diagrams: Dependencies and Reverse Dependencies. This package only has dependencies, so there is only one folder.

**Note** For structure pane folder definitions, see [“JBuilder UML diagrams defined” on page 11-7](#).

- 4 Open the Dependencies folder to see the packages, classes, and interfaces that the central package is dependent on. You can use the structure pane to navigate to other UML diagrams, as you'll see later.



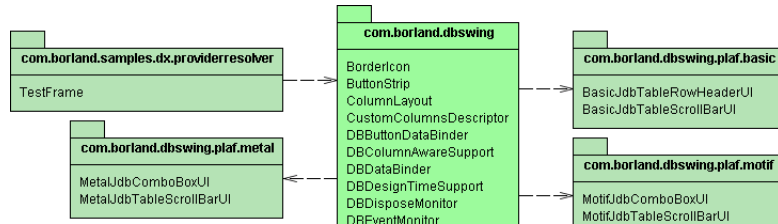
**Note** For structure pane icon definitions, see [“JBuilder structure pane and UML icons”](#) in online help.

In the following steps, navigate to a package that also has reverse dependencies: `com.borland.dbswing`.

- 1 Navigate to the `com.borland.dbswing` package, which is on the top left side of the diagram. There are two ways to do this:
  - Double-click the package name in the UML diagram.
  - Open a folder in the structure pane, right-click the package name, and choose Open.

This may take awhile to load. Now, the `com.borland.dbswing` package is the central package in the UML diagram. Notice that there are dashed lines pointing in both directions in this package diagram. The `com.borland.dbswing` package has *dependencies* and *reverse dependencies*. See [“JBuilder UML diagrams defined” on page 11-7](#) for definitions of these terms.

- 2 Look at the top left package, `com.borland.samples.dx.providerresolver`. This package has a dashed line pointing to the central package, instead of away from the central package. This is a reverse dependency. The `TestFrame` class in the `com.borland.samples.dx.providerresolver` package uses `dbSwing` components for the application UI, such as `JdbNavToolBar` and `JdbTable`. Since `TestFrame` has a dependency on `com.borland.dbswing`, then `com.borland.dbswing` has a reverse dependency on `com.borland.samples.dx.providerresolver`.



- 3 Look at the folders in the structure pane. There are two folders for this package: `Dependencies` and `Reverse Dependencies`.
- 4 Double-click the `Reverse Dependencies` folder in the structure pane and expand the `com.borland.samples.dx.providerresolver` package node to see that `TestFrame` is also listed here as a reverse dependency of `com.borland.dbswing`.
- 5 Double-click `TestFrame` in the UML diagram or the structure pane to see the `dbSwing` components it uses, as well as the other components from other packages. Now, you're viewing a UML class diagram with `TestFrame` as the central class and the components listed directly below the class name.
- 6 Choose `View | Hide All` to enlarge the UML browser and hide the project and structure panes.

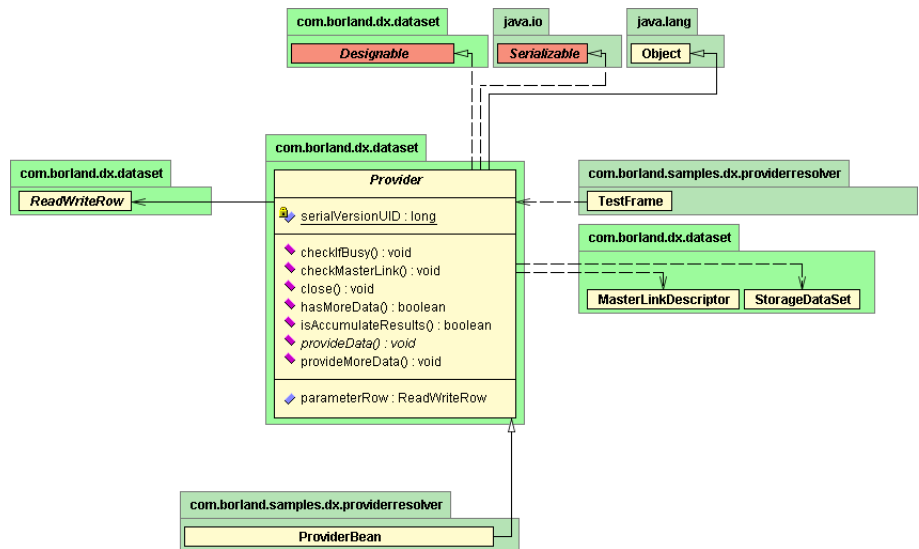
## Step 3: Viewing a UML class diagram

In this step, you'll navigate to another class diagram, in this case an abstract class called `Provider`. In UML diagrams, abstract classes are displayed in an italic font.

- 1 Scroll to the right and double-click the `Provider` class located in the `com.borland.dx.dataset` package, the second package on the right in the `TestFrame.java` UML diagram. Now, `Provider` is displayed as the central

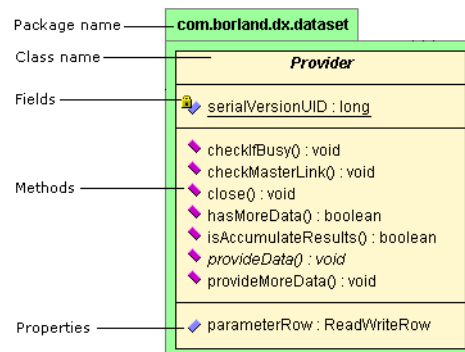
### Step 3: Viewing a UML class diagram

class in a UML class diagram. Notice that `Provider` is highlighted in the diagram indicating that it's selected.



Also, notice that all diagrams of the current package, `com.borland.dx.dataset`, display with bright green backgrounds by default, whereas other packages are a darker green.

The UML class diagram displays the class in the center of the diagram. Surrounding the class is the package with the package name in a tab at the top. The class itself is divided into several sections separated by horizontal lines in the following order:



- Class name at the top: abstract classes are italic.
- Fields and properties\*: static fields are underlined.

- Methods and getters\* and setters\*: `abstract` methods are italic, static methods are underlined.
- Properties\* (optional) at the bottom.

\*By default, properties are displayed in the bottom section. The Display Properties Separately option is set on the UML page of the IDE Options dialog box (Tools | IDE Options). If this option is turned off, properties are displayed in the appropriate sections with fields and methods. See [“Setting IDE Options” on page 11-19](#).

**Note**

Icons indicate whether a field, method, or property is `private`, `public`, or `protected`. For icon definitions, see “JBuilder structure pane and UML icons” in online help. You can also change these icons to generic UML visibility icons. See [“Setting IDE Options” on page 11-19](#).

Here are a few other things to notice in the diagram:

- Dependencies and reverse dependencies are on the right indicated with dashed lines.
- Associations are on the left indicated with solid lines.
- Extended (parent) classes and implemented interfaces are on the top. Extends relationships are represented by a solid line with a large triangle. Implements relationships use a dashed line with a large triangle.
- Extending classes are on the bottom. Extends relationships are represented by a solid line with a large triangle.
- Interfaces are represented by default with orange rectangles with their names in italics.
- Classes are represented by default with yellow rectangles.

See [“JBuilder UML diagrams defined” on page 11-7](#) for definitions of these terms.

- 2 Look at the methods listed in the third section of the central `Provider` class. One of the methods, `provideData()`, is an `abstract` method. Therefore, the method name is in an italic font.
- 3 Double-click the name of the `provideData()` method in the UML diagram to read the comments in the source code. This `abstract` method, which is inherited by the `ProviderBean` class at the bottom of the UML diagram, provides the data for a `DataSet`. Also, notice that this method is in the `abstract Provider` class.

- 4 Click the UML tab to return to the `Provider` UML class diagram. Below the central class are extending classes. `ProviderBean`, which provides the data to read into a `TableDataSet`, extends (inherits from) the abstract class `Provider`.
- 5 Right-click the `ProviderBean` class at the bottom of the diagram and choose **Go To Source** to examine some of its methods. See that it inherits the `provideData()` method from `Provider`'s abstract `provideData()` method and extends `Provider`.
- 6 Choose the UML tab to view the `ProviderBean` class diagram. Inheritance is represented in UML diagrams as a solid line with a large triangle that points to the parent. Because `ProviderBean` inherits from and extends `Provider`, there is a solid line with a large triangle that points from `ProviderBean` to `Provider` at the top of the diagram.
- 7 Position the mouse over the `provideData()` method in the `ProviderBean` class. A tool tip shows the method name with its parameters and return type. This is a convenient way to learn more about a method without leaving the diagram.

```
provideData(StorageDataSet, boolean): void
```
- 8 Double-click the `Provider` class at the top of the diagram or right-click and choose **Go To diagram** to return to its class diagram.

Now, look at the last section of the central `Provider` class diagram. By default, properties are listed separately. You can change the display of properties on the UML page of the IDE Options dialog box (Tools | IDE Options). A property exists when a method name following “is”, “get”, or “set” matches a field name. For example, in this diagram, `parameterRow` is a field with get and set methods. Therefore, `parameterRow` is a property.

- 1 Right-click the `parameterRow` field at the bottom of the `Provider` class in the UML diagram and choose **Go To Source**.
- 2 Choose **View | Show All**, so you can see the structure pane again.
- 3 Examine the methods in the structure pane and look for any “is”, “get”, or “set” methods. There are two methods with the same name as the `parameterRow` field: `getParameterRow()` and `setParameterRow(ReadWriteRow)`.

## Step 4: Adding references from libraries

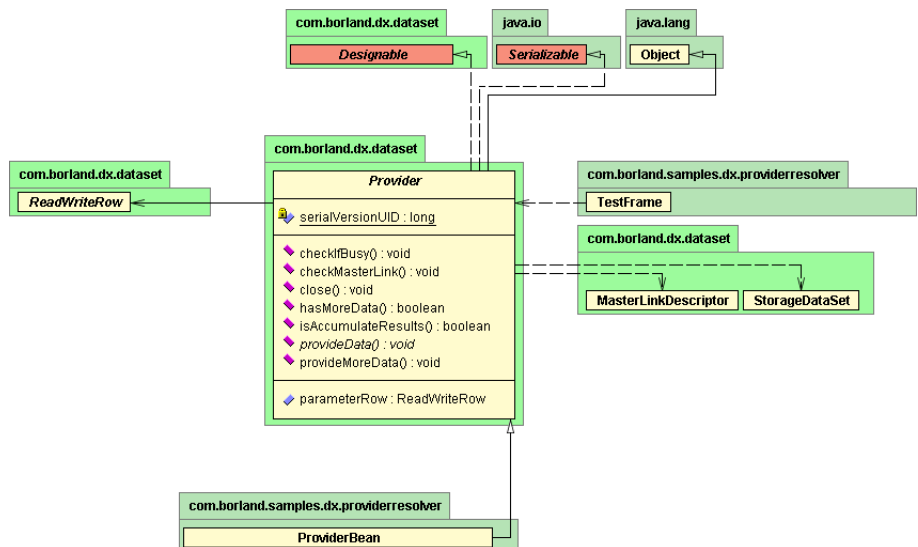
Sometimes, you might want to include references from project libraries to see a complete diagram of all relationships. By default, JBuilder's UML diagrams do not display the references from the project libraries. Typically, libraries provide services to the applications that are built upon them but don't know anything about their users. To show these relationships, you need to include references from the libraries.

When you choose the Include References From Project Library Class Files option on the General page of the Project Properties dialog box (Project | Project Properties), references from project library classes are also included in the UML diagram. After selecting this option, JBuilder automatically refreshes the diagram. For more information on this option, see ["Including references from project libraries" on page 11-18](#).

**Caution** If the libraries are large, the UML diagram may take some time to load. JBuilder must first load all the classes and the dependency information to determine the relationships.

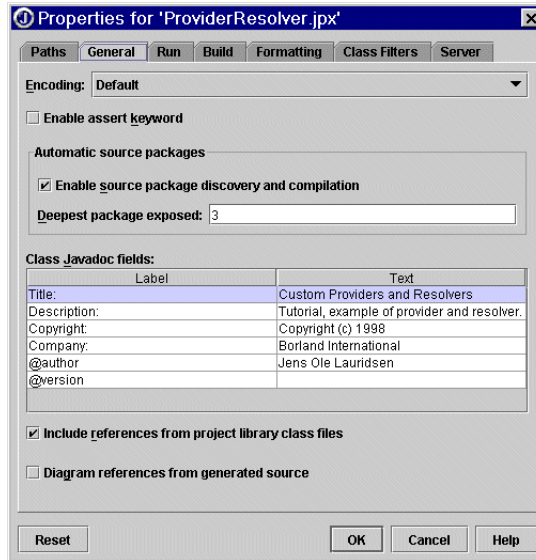
Before choosing the Include References From Project Library Class Files option, review the `Provider` class diagram. Then, change the project properties and see how the `Provider` diagram changes.

- 1 Choose the UML tab to return to the `Provider` UML class diagram. There is a package on the left containing the `ReadWriteRow` class and a package on the bottom of the diagram containing the `ProviderBean` class. When you add the references from the libraries, more packages and classes are added to the diagram in these areas.



#### Step 4: Adding references from libraries

- 2 Right-click `ProviderResolver.jpx` in the project pane and choose Properties to open the Project Properties dialog box.
- 3 Choose the General tab.
- 4 Check the Include References From Project Library Class Files option at the bottom of the page.



- 5 Click OK to close the dialog box. The UML diagram refreshes and now displays additional references from the project libraries.
- 6 Review the UML diagram and notice the additional packages and classes added to the left and bottom of the diagram. On the left, `StorageDataSet` has been added to the `com.borland.dx.dataset` package. On the bottom, another `com.borland.dx.sql.dataset` package with two classes is added. These classes at the bottom of the diagram extend `Provider`. The classes in the `com.borland.dx.sql.dataset` package are not





- 4 Select `StorageDataSet` in the structure pane to highlight it in the diagram. A reverse association is represented by a solid line that points from the association to the central class. `StorageDataSet` has a reference to `Provider`:

```
private Provider provider;
```
- 5 Right-click `StorageDataSet` in the diagram and choose Go To Source to navigate directly to the source code.
- 6 Do a search in the source code for `provider` (Search | Find) and choose Find All. Examine the search results in the message pane and see that there are many references to the `Provider` class, as well as `getProvider()` and `setProvider()` methods.
- 7 Right-click the Search Results tab in the message pane and choose Remove “Search Result” Tab to close the message pane.

## Step 5: Filtering UML diagrams

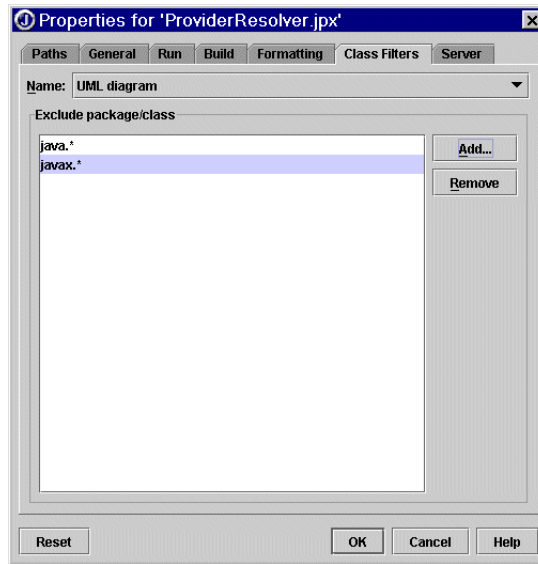
---

In some cases, you might want to remove packages and classes from your UML diagrams to simplify them. You can do this in the Project Properties dialog box (Project | Project Properties). For more information, see [“Setting project properties” on page 11-17](#).

The filtering of packages and classes is set on the Class Filters page of the Project Properties dialog box. The filtering determines what classes and packages are excluded from the UML diagrams in the project. Once you set the filters, you can enable and disable filtering from the UML browser’s context menu.

- 1 Choose the `com.borland.samples.dx.providerresolver` package file tab at the top of the content pane. Notice the many `java` and `javax` packages in the diagram. These packages also appear in the structure pane in the Dependencies folder.
- 2 Right-click `ProviderResolver.jpx` in the project pane and choose Properties to open the Project Properties dialog box.
- 3 Choose the Class Filters tab in the dialog box.
- 4 Exclude the `java` and `javax` packages from the `com.borland.samples.dx.providerresolver` package diagram as follows:
  - a Choose UML Diagram from the Name drop-down list.
  - b Choose the Add button to open the Select Package/Class dialog box.
  - c Choose `java` and click OK.
  - d Choose the Add button again.

- e Choose `javax` and click OK. Both packages now appear in the Exclude Package/Class list.



- f Click OK to close the dialog box.
- 5 Review the diagram and notice that the `java` and `javax` packages are now removed from the diagram. Only the `borland` packages remain in the diagram.
  - 6 Examine the structure pane to see that filtering doesn't remove the `java` and `javax` packages from the structure pane, although the packages do display in a lighter color to indicate that they aren't in the diagram.
  - 7 Right-click in the UML browser and notice that Enable Class Filtering is checked on the context menu.
  - 8 Turn off the filtering by choosing Enable Class Filtering to uncheck it. The packages reappear in the diagram.

#### Important

If you set filtering in the Project Properties dialog box, all of the diagrams in the project are filtered. Disabling filtering in one diagram does not disable it for all diagrams. If you navigate to another diagram in the project, filtering is still enabled. Once you close the file or package, the setting reverts back to the project-level setting.

Congratulations, you've completed the UML tutorial. There are many other features available in JBuilder's UML browser, such as:

- Refactoring code
- Saving and printing UML diagrams
- Viewing Javadoc
- Customizing UML diagrams

**See also**

- [Chapter 12, “Refactoring code symbols”](#)
- [“Creating images of UML diagrams” on page 11-20](#)
- [“Printing UML diagrams” on page 11-20](#)
- [“Viewing Javadoc” on page 14-20](#)
- [“Setting IDE Options” on page 11-19](#)

## Tutorial: Creating and running test cases and test suites

Unit testing is a feature of  
JBuilder Enterprise.

This tutorial shows you how to create a test case and a test suite to test existing code. The tutorial uses the ProviderResolver sample in `<jbuilder>/samples/DataExpress` as an example of an application under test. Before running this tutorial, make sure that you have installed the `samples` folder.

**Note** The tutorial in [Chapter 20, “Tutorial: Visualizing code with the UML browser”](#) also uses the ProviderResolver sample. If you plan to run both of these tutorials, it is recommended that you either run that tutorial first, or make a copy of the ProviderResolver sample before running this one since the modifications made in this tutorial may change some of the diagrams described in the UML tutorial.

This tutorial assumes you are familiar with Java, JUnit, and the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on JUnit, see the JUnit web site, <http://www.junit.org>. For more information on the JBuilder IDE, see “The JBuilder environment” in *Introducing JBuilder*.

**Note** You must have Build Target set to either Make or Rebuild in your current runtime configuration for the steps in this tutorial to work properly. Make is the default setting. For more information about runtime configurations, see [“Setting runtime configurations” on page 7-6](#).

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder’s ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see [“Documentation conventions” on page 1-4](#).

## Step 1: Opening an existing project

---

In this step, you open the `ProviderResolver` sample. For the purposes of this tutorial, `ProviderResolver` is the application under test.

- 1 Select **File | Open Project** to display the Open Project dialog box.
- 2 Click the **Samples** folder icon.
- 3 Open the **DataExpress** and **ProviderResolver** nodes in the tree.
- 4 Select `ProviderResolver.jpx` and click **OK**.

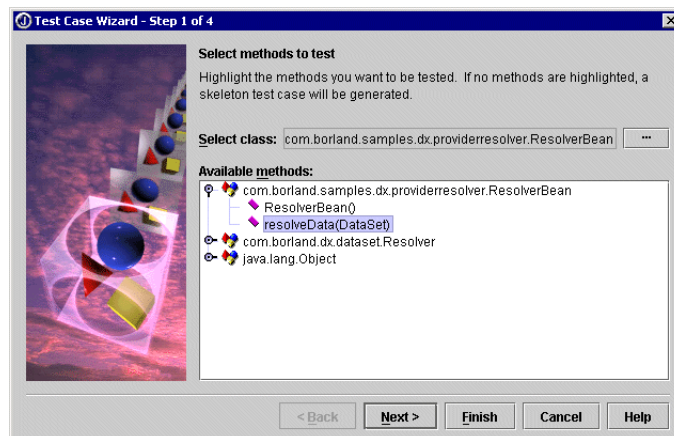
The `ProviderResolver` sample is now open.

## Step 2: Creating skeleton test cases

---

This step creates the skeleton of a test case using the Test Case wizard. The skeleton test case class will contain a test method for one of the methods in the `ResolverBean` class. Later you'll add a second test method. When writing unit tests in the real world, you may want to test more extensively, but for the purpose of this tutorial, you'll only implement two test methods.

- 1 Select **Project | Rebuild Project "ProviderResolver.jpx"**. This makes the methods of the project's classes available to the Test Case wizard.
- 2 Double-click `ResolverBean.java` in the project pane to open it in the editor.
- 3 Select **File | New** to display the object gallery.
- 4 Select **Test Case** from the Test page of the object gallery and click **OK**. The Test Case wizard is displayed with `ResolverBean` selected as the class to test.
- 5 Select the `resolveData(DataSet)` method of `ResolverBean`. The Test Case wizard looks like this:



6 Click Finish.

7 Open the `com.borland.samples.dx.providerresolver` package node in the project pane to see the test case that has been created. It's called `TestResolverBean.java`. Double-click this file to open it in the editor.

**Note** If the package node is not available, set the Enable Source Package Discovery And Compilation option on the General page of the Project Properties dialog box (Project | Project Properties).

## Step 3: Implementing a test method that throws an expected exception

---

Sometimes it is useful to write a test case to verify that an expected exception is thrown. Your test code should be specific enough to determine whether the exception that is thrown is the same exception that's expected. The test should fail if another exception is thrown.

In this step and the next one, you'll fill in the skeleton test case by writing test code. This step implements the `testResolveData()` method in `TestResolverBean`. To implement the method:

1 Replace the body of the `testResolveData()` method with the following code:

```
resolverBean = new ResolverBean();
com.borland.dx.dataset.DataSet dataSetView1=
    new com.borland.dx.dataset.StorageDataSet();

try{
    resolverBean.resolveData(dataSetView1);
    fail("failed: resolveData() did not throw an exception");
}
catch(com.borland.dx.dataset.DataSetException e){
    System.out.println("TestResolverBean.testResolveData(): success");
}
catch(Exception e){
    System.err.println("Exception thrown: "+e);
    fail("wrong exception: " + e.getClass().toString());
}
```



2 Click the Save All button on the toolbar, or select File | Save All.

3 Run the test now by right-clicking `TestResolverBean.java` in the project pane and selecting Run Test Using Defaults from the context menu. When the test finishes running, the progress bar of the test runner is green and green check mark icons are displayed next to the names of the test class and the test case in the test runner to indicate that the test passed.

The test code throws a `DataSetException` as expected because the data set isn't open. When the expected `DataSetException` is thrown, the exception is caught and the test passes because there was no assertion failure or error. If another class of exception were thrown, it would be caught by the second `catch` clause which calls the `fail()` method to ensure the test fails when the wrong exception is thrown.

You also added another line of test code after the line in the `try` block that is expected to throw an exception. If no exception is thrown, this line executes. The call to `fail()` in this line of code causes the test to fail and the string which is passed to it explains the failure. Of course, this new line of code can only be executed if an exception is not thrown.

## Viewing the test failure output

---

To demonstrate what a failure looks like in the test runner:

- 1 Comment out the following line of code in the `try` block:

```
resolverBean.resolveData(dataSetView1);
```



- 2 Click the Save All button on the toolbar, or select File | Save All.

- 3 Run the test now by right-clicking `TestResolverBean.java` in the project pane and selecting Run Test Using Defaults from the context menu. The progress bar turns red to indicate there has been at least one test failure. The Test Failures tab is displayed. A failure is listed for the `testResolveData()` method. This is indicated by a red X icon.



- 4 Click the `testResolveData()` node in the Test Failures page. This reveals the following output:

```
junit.framework.AssertionFailedError: failed: resolveData() did not  
throw an exception
```

```
...(Click for full stack trace)...
```

```
at com.borland.samples.dx.providerresolver.TestResolverBean.testResolveData  
(TestResolverBean.java:26)
```

```
...
```

Note how the string passed to the `fail()` method was carefully chosen to provide useful information in the event of a failure. This is a good objective to keep in mind when writing your tests.

## Fixing the test so it passes

---

You commented out a line of code that is necessary for the test to pass. The purpose of this was to see what a failure looks like in the test runner. To make the test pass again:



- 1 Uncomment the line of code that calls `resolverBean.resolveData(dataSetView1)`.



- 2 Click the Save All button on the toolbar, or select File | Save All.

If you run the test again at this point, it should pass. You can try this now if you like.

## Step 4: Writing a second test method

---

In this step you will write a method to test the value of one of the constants defined in the interface `DataLayout`, which is implemented by `ResolverBean`. This test verifies that the value of the constant is being set correctly. To do this:

- 1 Add the following method to the `TestResolverBean` class:

```
public void testConstant() {
    resolverBean = new ResolverBean();
    assertEquals(6, resolverBean.COLUMN_COUNT);
}
```



- 2 Click the Save All button on the toolbar, or select File | Save All.

The code you just added tests the value of the `COLUMN_COUNT` constant in the `DataLayout` interface to make sure it matches the expected value. In order to save time in this tutorial, only this one constant is tested, but it should give you an idea of some possible tests you could write to verify expected values. If you run the test now, it should pass since the value is being set correctly.

## Step 5: Creating a test suite

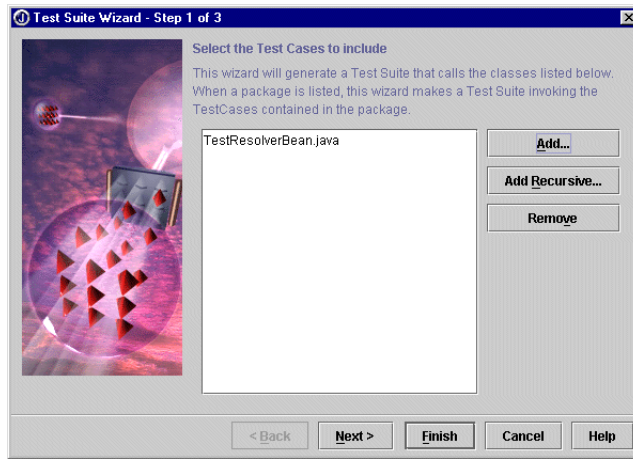
---

A test suite is a collection of tests that should be run as a group. In this step, you will create a test suite using the Test Suite wizard. This test suite will call `TestResolverBean`. Normally, a test suite calls more than one test case, but to save time in this tutorial, you have only created one test case. If you had more than one test case, the process for creating a suite that calls several test cases is the same.

- 1 Select File | New from the menu.
- 2 Select Test Suite from the Test page of the object gallery and click OK.
- 3 Select `TestResolverBean.java` as the test case to include in the test suite. Use the Add button to add it if necessary. If you had other test cases,

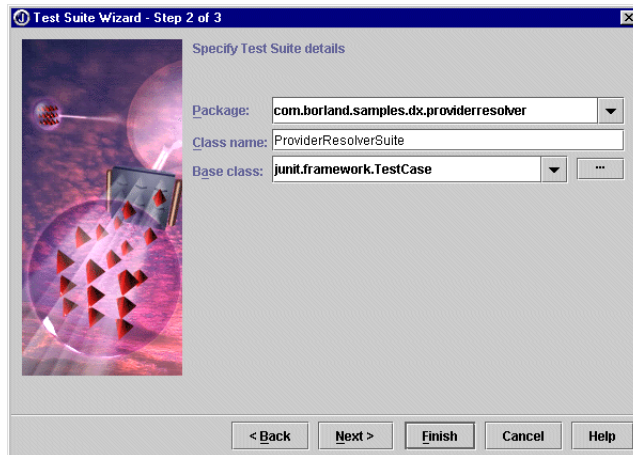
## Step 5: Creating a test suite

you could also add them at this stage. The Test Suite wizard looks like this:



4 Click Next.

5 Type `ProviderResolverSuite` as the Class Name. Now the Test Suite wizard looks like this:



6 Click Finish. A `ProviderResolverSuite.java` file is added to your project.

7 Double-click `ProviderResolverSuite.java` in the `com.borland.samples.dx.providerresolver` package in the project pane to open it. Note the following line of code in the `suite()` method:

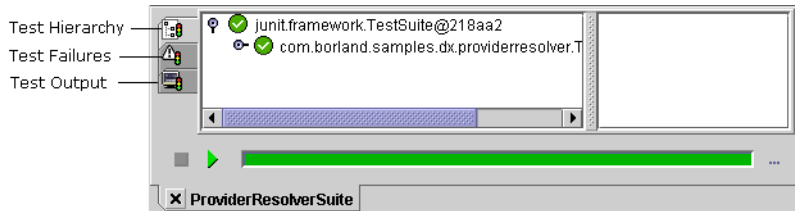
```
suite.addTestSuite(  
    com.borland.samples.dx.providerresolver.TestResolverBean.class);
```

If you later want to add other test cases to this suite, you would write a line of code similar to this one for each test case, substituting the class name of the new test case for `TestResolverBean`.

## Step 6: Running tests

In this step you will run the test suite you just created. The process for running a test suite is the same as for running a test case except that when you run a test suite, it automatically runs all the test cases in the suite. To run your test suite:

- 1 Right-click `ProviderResolverSuite.java` in the project pane and select Run Test from the context menu. The test runs.
- 2 Examine the output. JUnitRunner looks like this:



The top tab at the left of the JUnitRunner page in the message view is the Test Hierarchy view. Note that the tree shown in this view indicates that all the tests passed. The icons for the tests are all green check marks. Under the node for `ProviderResolverSuite`, you will see a subnode for your test case. If you had other test cases, there would be one subnode for each test case. Expand the test case node to see the individual tests. Click on a test case or individual test node to see the results for it.



- 3 Click the Test Failures tab. There is currently no output in this view. If there were failures, it would list them and show the output generated by their failed assertions.



- 4 Click the Test Output tab. This tab lists any output from the tests. The output of the test case you wrote in this tutorial is:

```
TestResolverBean.testResolveData(): success
```

Above this, the Test Output tab also shows the command that was used to run the tests.

You can also debug tests by right-clicking the test in the project pane and selecting Debug Test from the context menu. The test debugger works just like the regular debugger, so it is not discussed in this tutorial. The only difference is that when debugging a test the Test Hierarchy and Test Failures tabs from JUnitRunner are displayed in addition to the regular debugger UI. For more information on the debugger, see [Chapter 8, "Debugging Java programs."](#)

Congratulations! You've completed the tutorial on creating and running test cases and test suites. For more detailed information on unit testing, see [Chapter 13, "Unit testing."](#)



## Tutorial: Working with test fixtures

Unit testing is a feature of  
JBuilder Enterprise.

This tutorial illustrates how to use the JDBC Fixture wizard and the Comparison Fixture wizard to create fixtures for use in your unit tests. Fixtures are shared code that can be used by multiple unit test classes to perform routine tasks. This tutorial also shows you how to use the Comparison Fixture and the JDBC Fixture together in a test case.

This tutorial assumes you are familiar with Java, JUnit, JDataStore, and the JBuilder IDE. For more information on Java, see *Getting Started with Java*. For more information on JUnit, see the JUnit web site, <http://www.junit.org>. For more information on JDataStore, see *JDataStore Developer's Guide*. For more information on the JBuilder IDE, see "The JBuilder environment" in *Introducing JBuilder*.

**Note** You must have Build Target set to either Make or Rebuild in your current runtime configuration for the steps in this tutorial to work properly. Make is the default setting. For more information about runtime configurations, see "[Setting runtime configurations](#)" on page 7-6.

The Accessibility options section in the JBuilder Quick Tips contains tips on using JBuilder features to improve JBuilder's ease of use for people with disabilities.

For information on documentation conventions used in this tutorial and other JBuilder documentation, see "[Documentation conventions](#)" on page 1-4.

## Step 1: Creating a new project

---

- 1 Select File | New Project to display the Project wizard.
- 2 In the Name field, enter a project name, `fixturestutorial`.
- 3 Click Finish to close the Project wizard and create the project. You do not need to make any changes to the defaults on Steps 2 and 3 of the wizard.

A new project is created.

## Step 2: Creating a Data Module

---

In this step you will create a `DataModule` class to serve as the class under test. When writing unit tests in a real world situation, in most cases you will already have some code you want to test. For the purposes of this tutorial, we'll create some sample code and then test it in the steps that follow.

To create the Data Module:

- 1 Select File | New.
- 2 Select Data Module from the General page of the object gallery. Click OK. The Data Module wizard opens.
- 3 Accept the default Package and Class Name, uncheck Invoke Data Modeler, and click OK. A new Data Module is created.
- 4 Add the following import statement to the top of the `DataModule1.java` file, just after the line that reads `package fixturestutorial;`

```
import com.borland.dx.sql.dataset.*;
```

- 5 Add the following line of code just after the line that reads `private static DataModule1 myDM;`

```
Database database1 = new Database();
```

- 6 Add the following line of code to the `jbInit()` method, where `<drive>` and `<jbuilder>` are the actual drive and directory location of your JBuilder installation:

```
database1.setConnection(new ConnectionDescriptor  
("jdbc:borland:dslocal:<drive>:\\<jbuilder>\\samples\\JDataStore\\  
datastores\\employee.jds",  
"user", "", false, "com.borland.datastore.jdbc.DataStoreDriver"));
```

**7** Add the following method to the Data Module:

```
public Database getDatabase1() {
    return database1;
}
```

**8** Select Project | Rebuild Project “fixturetutorial.jpx”.

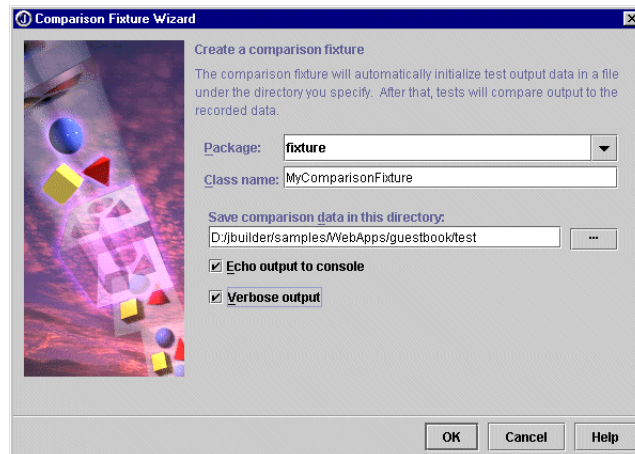
You’ve completed the Data Module. In the following steps, you’ll create test fixtures and a test case to test this Data Module.

## Step 3: Creating a comparison fixture

---

The Comparison Fixture wizard generates a fixture which is useful for recording test results and comparing them to previous test results. A Comparison Fixture extends `com.borland.jbuilder.unittest.TestRecorder`. To create a Comparison Fixture:

- 1** Select File | New from the menu.
- 2** Click the Test tab of the object gallery. Select Comparison Fixture and click OK. The Comparison Fixture wizard opens.
- 3** Accept the default in the Package field.
- 4** Enter `MyComparisonFixture` as the Class Name.
- 5** Accept the default for the comparison data directory location.
- 6** Check Echo Output To Console and Verbose Output. The Comparison Fixture wizard looks like this:



- 7 Click OK. A Comparison Fixture class called `MyComparisonFixture` is created. Expand the `fixturestutorial` node in the project pane to see it.

**Note** If the package node is not available, set the Enable Source Package Discovery And Compilation option on the General page of the Project Properties dialog box (Project | Project Properties).

- 8 Double-click `MyComparisonFixture.java` in the project pane to open it in the editor (it's in the `fixturestutorial` package). Note the following line of code:

```
super.setMode(UPDATE);
```

This line of code sets the output mode for the Comparison Fixture. Here are the possible values of the constants passed to `setMode()`:

- `UPDATE` - The comparison fixture compares new output to an existing output file, or creates the output file if it does not exist and records output to it.
- `COMPARE` - The comparison fixture always compares new output to the output that already exists.
- `RECORD` - The comparison fixture records all output, overwriting any previous output existing in the output file.
- `OFF` - The comparison fixture is disabled.

**Tip** If an existing output file contains incorrect data, set the output mode to `RECORD` after fixing the problem. Once you have recorded the desired output, set the mode back to `UPDATE`.

## Step 4: Creating a JDBC fixture

---

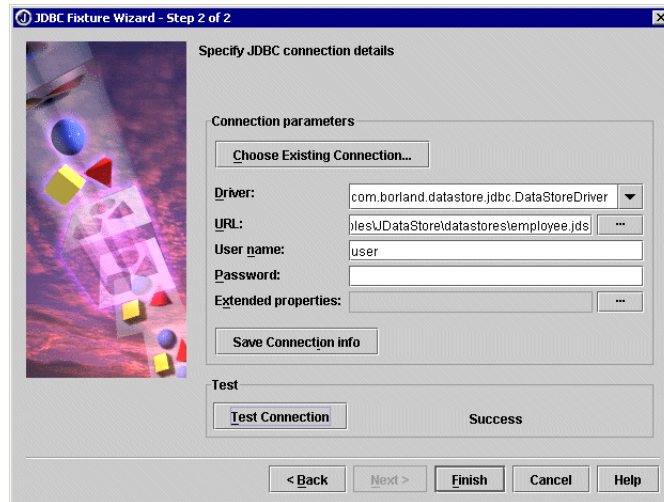
The JDBC Fixture wizard generates a fixture which is useful for managing connections to JDBC data sources.

To create a JDBC fixture:

- 1 Select File | New from the menu.
- 2 Click the Test tab of the object gallery. Select JDBC Fixture and click OK. The JDBC Fixture wizard opens.
- 3 Accept the defaults for Package, Class Name, and Base Class and click Next.
- 4 Select the following driver: `com.borland.datastore.jdbc.DataStoreDriver`
- 5 Enter or browse to the following URL: `jdbc:borland:dslocal:<drive>\<jbuilder>\samples\JDataStore\datastores\employee.jds` (where `<drive>` and `<jbuilder>` are replaced with your actual JBuilder location.)
- 6 Enter `user` for the User Name.



- 7 Click Test Connection. You should see a Success message to the right of the Test Connection button. The JDBC Fixture wizard looks like this:



If the connection fails, it may be because you don't have the correct JDataStore license information in the JDataStore License Manager. The JDataStore License Manager is available from the File menu of JDataStore Explorer.

- 8 Click Finish. A JDBC Fixture class called `JdbcFixture1` is created.
- 9 Double-click `JdbcFixture1.java` in the project pane to open it in the editor. Notice the `setUp()` and `tearDown()` methods in this fixture. In the next step, you will use these methods to run SQL scripts to manage data that could be used in your tests.

## Step 5: Modifying the JDBC Fixture to run SQL scripts

In this step you will modify the `setUp()` and `tearDown()` methods of the JDBC Fixture to make them run SQL scripts that automatically create test data before the tests are run and delete the test data when the tests are finished. To do this:

- 1 Make sure `JdbcFixture1.java` is open in the editor.
- 2 Add the String variables shown in **bold** to the fixture:

```
public class JdbcFixture1 extends JdbcFixture {

    String createSQL =
        "create table TESTTABLE (i int, j int);"+
        "insert into TESTTABLE values(1, 2);"+
        "insert into TESTTABLE values(2, 3);"+
```

## Step 6: Creating a test case using test fixtures

```
"insert into TESTTABLE values(3, 4);" +  
"insert into TESTTABLE values(4, 5);";
```

```
String deleteSQL =  
"drop table TESTTABLE;";
```

These `String` variables contain SQL statements which will be used to manage test data.

### 3 Add the code shown in **bold** to the `setUp()` method:

```
public void setUp() {  
    super.setUp();  
  
    Connection con = getConnection();  
    if(con != null)  
        runSqlBuffer(new StringBuffer(createSQL), true);
```

This code gets a connection to the `JDataStore` and runs the SQL script to create the test table.

### 4 Add the code shown in **bold** to the `tearDown()` method:

```
Connection con = getConnection();  
if(con != null)  
    runSqlBuffer(new StringBuffer(deleteSQL), true);  
super.tearDown();
```

This code gets a connection to the `JDataStore` and runs the SQL script to delete the test table.

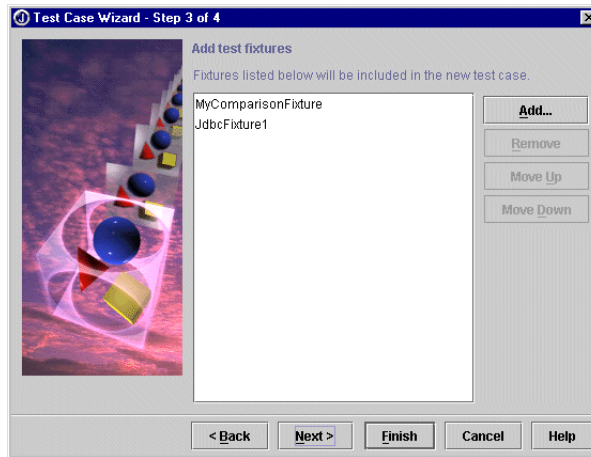
## Step 6: Creating a test case using test fixtures

---

In this step, you will use the Test Case wizard to include Comparison Fixture and JDBC Fixture in a test case.

- 1 Select Project | Rebuild Project “`fixturestutorial.jpj`”. This makes the methods of the project’s classes available to the Test Case wizard.
- 2 Double-click `DataModule1.java` to open it in the editor.
- 3 Select File | New from the menu.
- 4 Click the Test tab of the object gallery. Select Test Case and click OK. The Test Case wizard opens.
- 5 Accept `fixturestutorial.DataModule1` as the class to test. Don’t select any methods.
- 6 Click Next.
- 7 Accept the default class details in Step 2 of the wizard and click Next.

- 8 Use the Add button to add `MyComparisonFixture` and `JdbcFixture1` to the list of selected test fixtures, if they're not already there. The Test Case wizard looks like this:



- 9 Click Finish. A new test case called `TestDataModule1` is added to the project. Open the `fixturestutorial` node in the project pane to see it.

## Step 7: Implementing the test case

In this step, you'll write the code which calls the two test fixtures to use them in a test case.

- 1 Double-click `TestDataModule1.java` in the project pane to open it in the editor. The test case instantiates the fixtures and calls their `setUp()` and `tearDown()` methods.
- 2 Add the following line of code to the `import` statements at the top of `TestDataModule1.java`:

```
import java.sql.*;
```

- 3 Add the following method to the body of the `TestDataModule1.java` file:

```
public void testQuery() throws Exception{
    DataModule1 dm = new DataModule1();
    Connection con = dm.getDatabase1().getJdbcConnection();
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT * FROM TESTTABLE");
    jdbcFixture1.dumpResultSet(rs, myComparisonFixture);
    dm.getDatabase1().closeConnection();
}
```

This method does the following:

- Instantiates `DataModule1`.
- Gets a `Connection` using the `getJdbcConnection()` method of the `DataExpress Database` object used in `DataModule1`.
- Calls the `Connection.createStatement()` method in preparation for executing a SQL query.
- Executes the query, storing the result to a `ResultSet` object.
- Uses the JDBCTest's `dumpResultSet()` method to dump the result set to the `Comparison Fixture`. The `dumpResultSet()` method gets passed a `ResultSet` and a `Writer` as parameters. A `Comparison Fixture` can be used as the `Writer` because it extends `Writer`.
- Calls the `closeConnection()` method of the `Database` object to make sure the connection to the data source is closed.

## Step 8: Adding a required library

---

Before the test can run, you need to add the `JDataStore` library to your project. To do this:

- 1 Select **Project | Project Properties**.
- 2 Select the **Required Libraries** tab on the **Paths** page of the **Project Properties** dialog box.
- 3 Click the **Add** button.
- 4 Select the `JDataStore` library from the list and click **OK**.
- 5 Click **OK** to close the **Project Properties** dialog box.

## Step 9: Running the test case

---

In this step you will run the test case.

- 1 Right-click `TestDataModule1.java` in the project pane and select **Run Test Using Defaults** from the menu. The test runs. When the test is run, the following things happen:
  - The test runner instantiates `TestDataModule1`.
  - `TestDataModule1.setUp()` is called, which in turn calls the `setUp()` methods of the two fixtures in the proper order.

- The `testQuery()` method is called. Output from the comparison fixture is recorded to a data file in the same source directory where `TestDataModule1.java` is located, the `test/fixturestutorial` subdirectory of your project directory.
- The `tearDown()` method is called, which in turn calls the `tearDown()` methods of the two fixtures in the proper order.

Congratulations! You've completed the test fixtures tutorial. For more detailed information on unit testing, see [Chapter 13, "Unit testing."](#)





## Creating configuration files for native executables

Configuration files provide flexibility and customization in launching applications and utilities. For example, they can be used to pass parameters to the Java Virtual Machine (VM), pass parameters to OpenTools, debug OpenTools, and customize the launching of applications.

The *launcher* for the application contains the appropriate executables, shell scripts, and/or desktop icons and is used to start the application. Before it launches, it looks for a *configuration file* for additional instructions. A configuration file is a text file containing a list of case-sensitive directives to be executed before launching an executable.

After finding the configuration file, the launcher processes each line of text sequentially before the Java VM loads. If the launcher doesn't find a configuration file, it opens itself as a file and finds the configuration file stored inside as a zip comment. The launcher reports an error and terminates if it can't read the configuration file, if any line contains an unrecognized directive, if the `mainclass` directive is missing, or if the Java VM fails to start. An error is also reported if the `javapath` directive is omitted and a default Java VM location can't be determined.

The directives in the configuration file might specify the main class, add JAR files, pass parameters to the VM, add a path entry to the Java classpath, and so on. Configuration files can also include other configuration files. For example, the JBuilder launcher refers to the `jbuilder.config` file, which then uses the `include` directive to refer to `jdk.config` as shown in the following example.

If you're creating executables with the Archive Builder or the Native Executable Builder, you can choose to override the default configuration

file created by these wizards. For more information, see [“Deploying with the Archive Builder” on page 15-17](#) and [“Creating executables with the Native Executable Builder” on page 15-29](#).

### Sample configuration file

```
# Read the shared JDK definition
include jdk.config

# Tune this VM to provide enough headroom to work on large
# applications
vmparam -Xms32m
vmparam -Xmx128m

# Put the Light AWT wrapper on the boot path
addbootpath ../lib/lawt.jar
addbootpath ../lib/TabbedPaneFix.jar

# Add all JAR files located in the patch, lib and lib/ext directory
addjars ../patch
addjars ../lib
addjars ../lib/ext

# Include the Servlet 2.3 API from Tomcat 4 in the IDE classpath
addpath ../jakarta-tomcat-4.0.3/common/lib/servlet.jar

# Activate the shell integration
socket 8888

# Add all the configuration files located in the lib/ext directory
includedir ../lib/ext

# JBuilder needs to have access to the environment
exportenv

# Start JBuilder using the main class
mainclass com.borland.jbuilder.JBuilder
```

When you create native executables for your applications with the JBuilder Archive Builder and Native Executable Builder, you can customize the launching behavior of the application with a configuration file on the Executables page. For example, you might want to pass certain VM and runtime parameters to your application before it launches. For more information on creating native executables, see [“Creating executables with the Native Executable Builder” on page 15-29](#).



# Starting the VM

---

The command to start the VM at the command line has the following form:

```
<javapath> [-Xbootclasspath/p:<bootpath>]
[-classpath <classpath>] <vmparams> <mainclass> {params}
```

The elements in <angle brackets> represent values derived from the configuration file and [square brackets] represent optional parts of the command line that are omitted if relevant path elements aren't defined in the configuration file. The {params} component is by default a duplicate of the launcher's command-line parameters, but may be altered by some configuration file directives.

## Configuration file requirements

---

The configuration file must meet certain specifications for it to be parsed correctly.

### File type and location

---

The configuration file must be a plain, unformatted text file. It should be in the same directory as the launcher and have the same name with a .config extension. For example, bcj.exe, located in the JBuilder bin directory, has a configuration file named bcj.config. If the launcher doesn't find a configuration file, it opens itself as a file and finds the .config file stored inside.

### Blank lines and comments

---

Blank lines and lines beginning with the # character are ignored to allow the configuration file to be structured and documented.

### Path conventions

---

All paths in the configuration file may be relative or absolute. Relative paths are relative to the directory containing the configuration file and the launcher. You can use “..” in the path to move to the parent directory. All path separators must be forward slashes, regardless of the standard path separator for the local platform.

## Directives

---

The following directives can be specified in the configuration file. Some are required while others are optional.

### javapath

---

The `javapath` directive provides the exact location and name of the Java interpreter. Whether this launcher is an executable or a shared library depends on the launcher. For example, a Win32 launcher might require one or the other of the following:

```
javapath ../jdk14/bin/java.exe
javapath ../jdk14/jre/bin/client/jvm.dll
```

If a `javapath` directive isn't found, the launcher attempts to determine a default Java VM location in the appropriate platform-dependent manner.

The platform-specific executable file looks for the installed JDK in the following location:

- Windows: Registry.
- Linux/Solaris: `JAVA_HOME` environment variable and the user's path.
- Mac OS X: pre-defined location for the JDK.

**Note** You can override this default behavior by specifying the location of the JDK in a custom configuration file. Then the executable file will look in the specified location. For more information on configuration files, see the next step.

The `javapath` directive is required if the JDK isn't installed in the usual location for the platform.

### mainclass

---

The `mainclass` directive, which is required, provides the fully qualified class name used to start the application. For example:

```
mainclass com.borland.jbuilder.JBuilder
```

### addpath

---

The `addpath` directive adds a single path to the class path used to start the application. For example:

```
addpath ../lib/jbuilder.jar
```

Note that the following additional rules apply:

- Paths that refer to a directory or file that doesn't exist aren't added.
- Paths already in the class path, boot path, and skip path aren't added.
- Paths that contain spaces are automatically placed between quotes when building the command line.

## addjars

---

The `addjars` directive adds all JAR files in the specified directory to the class path used to start the application. For example:

```
addjars ../lib
```

The same rules applied to the `addpath` directive apply to each of the paths added as a result of the `addjars` directive.

## addbootpath

---

The `addbootpath` directive adds a single path to the boot path used to start the Java VM. For example:

```
addbootpath ../lib/lawt.jar
```

Note that the following additional rules apply:

- Paths that refer to a directory or file that doesn't exist aren't added.
- Paths already in the class path are removed from the class path and then added to the boot path.
- Paths already in the boot path and the skip path aren't added.
- Paths that contain spaces are automatically placed between quotes when building the command line.

## addbootjars

---

The `addbootjars` directive adds all JAR files in the specified directory to the boot path used to start the application. For example:

```
addbootjars ../lib
```

The same rules applied to the `addbootpath` directive apply to each of the paths added as a result of the `addbootjars` directive.

## addskippath

---

The `addskippath` directive defines a single path that should never be added to the boot or class paths used to start the Java VM. This is especially useful for eliminating individual paths that would otherwise be added by `addjars` or `addbootjars`. For example:

```
addskippath ../lib/dbswing.jar
```

Note that the following additional rules apply:

- The path is removed from the class path if it's already been added.
- The path is removed from the boot path if it's already been added.

## vmparam

---

The `vmparam` directive provides parameters that are passed directly to the Java VM when it's started. For example, the following directive sets the minimum and maximum heap sizes to 8MB and 128MB respectively:

```
vmparam -Xms8m -Xmx128m
```

The effects of this directive are cumulative. Each subsequent occurrence adds to the set of parameters that are passed to the VM with spaces automatically inserted between parameters.

## include

---

The `include` directive causes the contents of the named file to be parsed before continuing with the current configuration file. This directive may be used to nest configuration files to an arbitrary number of levels. The launcher reports an error and terminates if the named file can't be read.

```
include jdk.config
```

As with all paths in the configuration file, the path may be relative or absolute. See [“Path conventions” on page A-3](#).

## includedir

---

The `includedir` directive causes all of the files with a `.config` extension in the specified directory to be processed as by the `include` directive.

```
includedir ../lib/ext
```

As with all paths in the configuration file, the path may be relative or absolute. See [“Path conventions” on page A-3](#).

## copyenv

---

The `copyenv` directive causes the contents of an environment variable to be exposed by defining a corresponding Java environment variable.

```
copyenv PROMPT
```

The Java variable has a prefix to avoid namespace collisions, so this directive defines a Java environment variable named `borland.copyenv.PROMPT` whose value is originally derived from the `PROMPT` environment variable.

## exportenv

---

The `exportenv` directive causes the contents of all system environment variables to be exposed by writing them to a temporary file. Each invocation of the launcher creates a unique file using the Java `.properties` file format to represent a complete name/value pair collection for all environment variables.

```
exportenv
```

The Java environment variable `borland.exportenv` is set to contain the file name that the environment has been written to. Once the Java VM terminates, it's the launcher's responsibility to delete the temporary file.

## addparam

---

The `addparam` directive appends a new parameter or set of parameters to the existing set of application parameters.

```
addparam <param>
```

## clearparams

---

The `clearparams` directive discards the existing set of application parameters. This directive typically is used in conjunction with subsequent `addparam` directives.

```
clearparams
```

## restartcode

---

The `restartcode` directive specifies an exit code for the Java process that should be interpreted as a request to restart the launch process. This directive is typically used by an application that needs to make changes to

its configuration, so the launcher must re-read configuration files as if this were a fresh launch.

`restartcode 22`

**Caution** Care must be used with this directive since it can easily lead to an endless cycle of VMs being started. For safety, a restart exit code of 0 is never interpreted as a restart request, even if an explicit `restartcode 0` directive is encountered.

## Optional all-in-one launcher support

---

To enable simple application distribution, launcher implementations can support two optional features:

- 1 Determining a default Java VM location in the absence of a `javavm` directive.
- 2 If there isn't an `<exename>.config` file and the launcher appears to match the format of a JAR file:
  - a Automatically adding the executable to the path as if by an explicit `addpath` directive
  - b Read the executable/JAR file's comment and interpret it as a configuration file

These two optional features together make it possible to create a standalone executable containing the launcher, Java code, and any necessary configuration information including the name of the main class. All it takes is concatenating the launcher executable with a JAR whose comment follows the configuration file format.

## Using the command-line tools

JBuilder includes the following command-line tools:

**bmj** and **bcj** are features  
of JBuilder SE and  
Enterprise

- JBuilder command-line interface
- Borland Make for Java (**bmj**)
- Borland Compiler for Java (**bcj**)

The JDK includes the following command-line tools:

- **javac** - the compiler for the Java programming language.
- **java** - the launcher for the Java applications.
- **jar** - manages the Java Archive (JAR) files.
- **javadoc** - extracts code comments and generates HTML documentation from those comments.
- **appletviewer** - allows you to run applets outside of the context of a web browser.
- **native2ascii** - converts a file of native encoded characters to one with Unicode escape sequences.

### See also

- Sun Tools documentation at <http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html>

## Setting the class path for command-line tools

---

The class path tells Java tools where to find classes that are not part of the Java platform. You can set the path to the classes with the **-classpath** option or by setting the `CLASSPATH` environment variable described in the following topic. The **-classpath** option temporarily overrides the `CLASSPATH` environment variable for the current command-line session. It's best to use **-classpath** as you can set it for each application, and it does not affect other applications.

Directories listed in the classpath are separated by colons on the UNIX platform and by semicolons on the Windows platform. You should always include the system classes at the end of the path. The classpath is also used to search for sources if no sourcepath is specified.

For more information on class paths, see the Java documentation on "Setting the classpath."

For more information on the JBuilder IDE and class paths, see ["How JBuilder constructs paths" on page 4-9](#) and ["Where are my files?" on page 4-12](#).

### Using the -classpath option

---

Use the **-classpath** option to temporarily set the path to your classes.

- **UNIX**

The **-classpath** option takes the following form:

```
% jdkTool -classpath path1:path2
```

- **Windows**

The **-classpath** option takes the following form:

```
C:>jdkTool -classpath path1;path2
```

### Setting the CLASSPATH environment variable for command-line tools

---

The **-classpath** command-line option only temporarily overrides the classpath and does not interfere with other applications. However, you can permanently set the `CLASSPATH` environment variable.

For more information on the **-classpath** option and the `CLASSPATH` environment variable, see the Java documentation, "Setting the classpath."



**UNIX: CLASSPATH environment variable**

To view the `CLASSPATH`,

- 1 Open a command-line shell window.
- 2 View the current `CLASSPATH` environment variable using the following command:

- in `csh` shell:

```
env
```

- in `sh` shell:

```
CLASSPATH
```

To set your `CLASSPATH` environment variable,

- 1 Open a command-line shell window.
- 2 Set the `CLASSPATH` environment variable using the following command-line format:

- in `csh` shell:

```
setenv CLASSPATH path1:path2
```

- in `sh` shell:

```
CLASSPATH = path1:path2
export CLASSPATH
```

To clear the `CLASSPATH`,

- 1 Open a command-line shell window.
- 2 clear the `CLASSPATH` environment variable using the following command-line format:

- in `csh` shell:

```
unsetenv CLASSPATH
```

- in `sh` shell:

```
unset CLASSPATH
```

**Windows: CLASSPATH environment variable**

To view the current `CLASSPATH`, use the `set` command.

```
C:> set
```

To set the `CLASSPATH` environment variable,

- 1 Open a DOS window.
- 2 Modify the `CLASSPATH` environment variable with the `set` command.

```
set CLASSPATH=path1;path2 ...
```

The paths must begin with the drive letter, for example `C:\`.

To clear your path, you can unset `CLASSPATH` as follows:

```
C:> set CLASSPATH=
```

This command unsets `CLASSPATH` for the current DOS session only. Be sure to delete or modify your startup settings to ensure that you have the right `CLASSPATH` settings in future sessions.

If the `CLASSPATH` variable is set at system startup, the place to look for it depends on your operating system.

- Windows NT: Choose Start | Settings | Control Panel | System to open the System Properties dialog box. Click the Environment tab and edit the `CLASSPATH` variable in the User Variables section.
- Windows 2000: Choose Start | Settings | Control Panel | System to open the System Properties dialog box. Click the Advanced tab and click the Environment Variables button to edit the `CLASSPATH` variable in the User Variables section. If you aren't logged on as administrator to the local computer, you can only change user variables.
- Windows XP: Choose Start | Settings | Control Panel | System to open the System Properties dialog box. Click the Advanced tab and click the Environment Variables button. If you aren't logged on as administrator to the local computer, you can only change user variables.

## JBuilder command-line interface

---

JBuilder has a command-line interface that includes such arguments as:

- Building projects
- Displaying configuration information
- Displaying the license manager
- Disabling the splash screen
- Enabling verbose debugging mode for OpenTools authors

**Note** These arguments vary by JBuilder edition.

JBuilder runs on its own launcher, which is an executable. The executable can pass arguments to JBuilder.

### Accessing a list of options

---

To access the list of arguments available in your edition of JBuilder, open a command-line window, navigate to the JBuilder `bin` directory and type `jbuilder -help`.

```
/<jbuilder>/bin> jbuilder -help
```

## JBuilder responds with a list of available arguments:

Available command-line arguments:

```
-build: Build JBuilder projects (not available in Personal)
-help: Display help on command line options
-info: Display configuration information
-license: Displays the license manager
-nosplash: Disable splash screen
-verbose: Display OpenTools loading diagnostics
```

To learn more about each argument, enter `jbuilder -help <argument_name>`, as shown in this example:

```
/<jbuilder>/bin> jbuilder -help info
```

## JBuilder responds with this information:

```
-info
  Displays system configuration information while loading.
```

You can also list a series of arguments, as shown in this example:

```
/<jbuilder>/bin> jbuilder -help info nosplash verbose
```

## Syntax

---

```
jbuilder [arguments]
```

## Options

---

- **-build** <args>

This is a feature of  
JBuilder SE and  
Enterprise.

Builds one or more JBuilder projects supplied as arguments. All settings are taken directly from the specified project files. Targets can be specified and are executed in the order listed. If a target isn't specified, make is the default target.

Among the **-build** arguments, projects are distinguished from targets by the `.jpx` or `.jpr` extension. Any argument that ends with `.jpx` or `.jpr` is assumed to be a project. Any argument that doesn't have a `.jpx` or `.jpr` extension is assumed to be a target name.

**Note**

If a project fails to complete, the remaining projects aren't built.

The command is in this form:

```
jbuilder -build <project1.jpx> [ [<target1> <target2> ...]
  [<project2.jpx> [<target3> ...] ] ... ]
```

For example:

```
jbuilder -build myproject.jpx rebuild
```

```
jbuilder -build myproject.jpx clean make myotherproject.jpx
```

**Note**

If the project is not in the current directory, include the complete path to the project file. For example, if the projects are located in the /user/username/ directory, the complete path would be:

```
jbuilder -build /user/username/myproject.jpx clean make  
/user/username/myotherproject.jpx
```

The command-line build process reports errors and warnings as text which can be redirected to another process or file. For example, on the Windows platform, you could redirect to a text file as follows:

```
jbuilder -build myproject.jpx > myproject.txt
```

The command-line build process returns a result code to indicate failure or success. If the build is successful, a zero value is returned. If the build fails, a non-zero value is returned. The actual value reported depends on the error.

In this example, a Windows .BAT file builds the project and echoes an error or success message:

```
rem This has to be in a .BAT file:  
jbuilder -build myproject.jpx  
if not errorlevel 0 echo ERROR  
if errorlevel 0 echo SUCCESS  
rem End of .BAT file
```

In addition, the command-line build process allows automation of build processes with other command-line tools. For example, you could execute more complicated tasks, such as build another project if the build is successful and then copy files or cancel the batch file if the build fails.

See your OS documentation for more information on what scripts or batch programs your system supports.

- **-help** <args>

Lists the JBuilder command-line arguments available.

When invoked without arguments, **-help** displays a list of recognized command-line arguments and a brief description of each. Invoking help with one or more arguments provides a more detailed description of the specified commands and their arguments.

```
-help <argument1> <argument2> <argument3>
```

Note that arguments must be typed without a leading hyphen.

- **-info**  
Displays system configuration information while loading.
- **-license**  
Displays the license manager instead of starting JBuilder.
- **-nosplash**  
Disables the splash screen that displays when JBuilder launches.
- **-verbose <args>**  
Displays OpenTools loading diagnostics.

## Borland Compiler for Java (bcj)

---

This is a feature of  
JBuilder SE and  
Enterprise

### Syntax

---

```
bcj [ options ] {file.java}
```

### Description

---

Borland Compiler for Java (**bcj**) compiles Java source code into Java bytecodes from the command line. **bcj** produces the Java program in the form of `.class` files containing bytecodes that are the machine code for the Java virtual machine. Compiling a source file produces a separate `.class` file for each class declaration or interface declaration. When you run the resulting Java program on a particular platform, such as Windows NT, the Java interpreter for that platform runs the bytecodes contained in the `.class` files.

**bcj** compiles the selected `.java` file and any files specified on the command line. **bcj** compiles the specified `.java` file, whether or not its `.class` file is outdated. An outdated `.class` file is one that was not generated by compiling the current version of its `.java` source file. Imported `.java` files that already have `.class` files will not be recompiled, even if their `.class` files are outdated; after using the **bcj** command, some imported classes might still have outdated `.class` files.

**bcj** does **not** check dependencies between files. For more information on **bmj** and smart dependencies checking, see [“Borland Make for Java \(bmj\)” on page B-12](#) and [“Smart dependencies checking” on page 5-2](#).

To see the syntax and list of options at the command line, enter the **bcj** command with no arguments from the `<jbuilder>/bin`.

You might need to use the **-classpath** option or set the `CLASSPATH` environment variable for the command line, so the required classes are found.

### See also

- [“Compiling from the command line” on page 5-9](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page B-2](#)
- [“Setting the classpath” in the Java Tools documentation](#)

## Options

---

**Note** Directories listed in paths are separated by colons on the UNIX platform and by semicolons on the Windows platform. The following examples represent the UNIX platform.

- **-classpath** path

The path used to find classes. Overrides the default or the `CLASSPATH` environment variable. You should always include the outpath at the beginning of the path. The classpath is also used to search for sources if no sourcepath is specified. For example:

```
bcj -classpath jbproject/testing/classes/test3:  
jbproject/project1/classes tester.java
```

- **-d** directory

The root directory of the class (destination) file hierarchy. Also referred to as the “outpath”.

For example, the following statement:

```
bcj -d jbproject/project1/classes tester.java
```

causes the class files for the classes defined in the `tester.java` source file to be saved in the directory `jbproject/project1/classes/test/test3`, assuming that `tester.java` contains the following package statement:

```
package test.test3;
```

Files are read from the class path and written to the destination directory. The destination directory can be part of the class path. The default destination matches the package structure in the source files and starts from the root directory of the source.

- **-deprecation**

Displays all deprecated classes, methods, properties, events, and variables used in the API.

If a warning is displayed when compiling, indicating that some deprecated APIs were used, you can turn this option on to see all deprecated APIs.

- **-encoding** *name*

You can specify a file-encoding name (or `codepage` name) to control how the compiler interprets characters beyond the ASCII character set. The default is to use the default native-encoding converter for the platform. For more information, see [“Specifying a native encoding for the compiler” on page 16-12](#).

For example, the following statement:

```
bcj -encoding EUC_JP tester.java
```

compiles `tester.java`. All source files are interpreted as being encoded in the `EUC_JP` character set, which is the character set typically used for Japanese UNIX environments. You can specify any encoding that is supported by the Java 2 platform. A list of the valid encodings is available at <http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html#intl>.

- **-exclude** *classname*

Excludes all calls to `static void` methods in the selected `.class` file from a compile. This also excludes the evaluation of the parameters passed to those methods.

For example, excluding class `A` removes all calls to `static void` methods of `A` from `OTHER` classes.

- **-g**

Generates all debugging information in the class file, including local variables. By default, only line numbers and source file information is generated.

- **-g:none**

Does not generate any debugging information.

- **-g:{keyword list}**

Generates some types of debugging information. The keyword list is a comma separated list of keywords, for example: `bcj -g:source,lines`

Keywords include:

- **source**: source file debugging information.
- **lines**: line number debugging information.
- **vars**: local variable debugging information.

- **-nowarn**

Compiles without displaying warnings.

- **-obfuscate**

Obfuscation makes your programs less vulnerable to reverse engineering. After decompiling your obfuscated code, the generated source code contains altered symbol names for private symbols.

- **-quiet**

Compiles without displaying any messages.

- **-source** version

Enables support for compiling source code containing assertions.

- When the version is set to 1.4, this command-line option enables the `assert` keyword and the JDK 1.4 assertion facility.
- When the version is set to 1.3, the compiler does not support assertions.
- If the `-source` option isn't used, the compiler defaults to the 1.3 behavior.

### See also

- “Assertion Facility” at <http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>
- **-sourcepath** path

The path used to find sources. If no source path is specified, the classpath is used to find the sources.

Similar to the classpath, the sourcepath must point to the root of the package directory tree, and not directly to the directory of the sources.

For example, to compile `tester.java`, which contains the package statement `package test.test3;` and is located in `jbproject/project1/src/test/test3`, you must set the source path to `jbproject/project1/src` and not to `jbproject/project1/src/test/test3`.

You can then type the following:

```
bcj -sourcepath jbproject/project1/src
    jbproject/project1/src/test/test3/tester.java
    -d jbproject/project1/classes
```

- **-verbose**

This option gives more information about compiling, such as which class files are loaded from where in the classpath, including:

- Which source files are being compiled.
- Which classes are being loaded.
- Which classes are generated.



## Cross-compilation options

---

**bcj** supports cross-compilation, where classes are compiled against a bootstrap and extension classes of a different Java platform. Use **-bootclasspath** and **-extdirs** when cross-compiling.

- **-target** *version*

Restricts the class files to work only on a specific VM version.

**bcj** supports:

- 1.1 - Generates class files to run on 1.1 and VMs in the Java 2 SDK. When you select this as the target VM, your class files can be loaded by any VM.
- 1.2 - Generates class files to run **only** on VMs in the Java 2 SDK, v 1.2 and later, but **won't** run on 1.1 VMs. This is the default.
- 1.3 - Generates class files to run **only** on VMs in the Java 2 SDK, v 1.3 and later, but **won't** run on 1.1 or 1.2 VMs.
- 1.4 - Generates class files to run **only** on VMs in the Java 2 SDK, v 1.4 and later, but **won't** run on 1.1, 1.2, or 1.3 VMs.

- **-bootclasspath** *bootclasspath*

Cross-compile against the specified boot classes. Entries can be directories, JAR archives, or ZIP archives.

- **-extdirs** *directories*

Cross-compile against the specified extension directories. Each JAR file in the specified directories is searched for class files.

## VM options

---

- **-J***option*

Passes options to the `java` launcher called by **bcj**. For example, **-J-Xms48m** sets the startup memory to 48 megabytes. It is a common convention for **-J** to pass options to the underlying VM executing applications written in Java.

Note that `CLASSPATH`, **-classpath**, **-bootclasspath**, and **-extdirs** do not specify the classes used to run **bcj**. If you do need to do this, use the **-J** option to pass through options to the **bcj** launcher.

# Borland Make for Java (bmj)

---

This is a feature of  
JBuilder SE and  
Enterprise

## Syntax

---

```
bmj [options] rootClasses
```

## Description

---

Borland Make for Java (**bmj**), which is the default compiler in the IDE and a command-line compiler, compiles Java source code into Java bytecodes and checks for dependencies to determine which files actually need to be recompiled. **bmj** produces the Java program in the form of `.class` files containing bytecodes that are the machine code for the Java virtual machine. Compiling a source file produces a separate `.class` file for each class declaration or interface declaration. When you run the resulting Java program on a particular platform such as Windows NT, the Java interpreter for that platform runs the bytecodes contained in the `.class` files.

**bmj** looks for dependency files on the classpath. If you specify a set of sources, some or all of those sources might not be recompiled. For example, the class files might be determined to be up to date if they have been saved but not edited since the last compile. You can force recompilation using the **-rebuild** option.

To check a set (or graph) of interdependent classes, it is sufficient to call **bmj** on the root class (or multiple root classes, if one is not under the other). You can specify root classes using class names, package names, names of sources that declare classes, or a combination.

You might need to set the `CLASSPATH` environment variable for the command line, so the required classes are found.

To see the syntax and list of options at the command line, enter the `bmj` command with no arguments from the `<jbuilder>/bin` directory. Many of these options can also be specified in the JBuilder IDE on the General page and the Build page of Project Properties (Project | Project Properties).

### See also

- [“Smart dependencies checking” on page 5-2](#)
- [“Compiling from the command line” on page 5-9](#)
- [“Setting the CLASSPATH environment variable for command-line tools” on page B-2](#)
- [“Setting the classpath” in the Java Tools documentation](#)
- [“Borland Compiler for Java \(bcj\)” on page B-7](#)

## Options

---

**Note** Directories listed in paths are separated by colons on the UNIX platform and by semicolons on the Windows platform. The following examples represent the UNIX platform.

- **-classpath** *path*

The path used to find classes and dependency files. Overrides the default or the `CLASSPATH` environment variable. You should always include the outpath at the beginning of the path. The outpath is the root directory of the class file hierarchy. The classpath is also used to search for sources if no sourcepath is specified.

For example:

```
bmj -classpath jbproject/testing/classes/test3:
jbproject/project1/classes tester.java
```

- **-d** *directory*

The root directory of the class (destination) file hierarchy. Also referred to as the outpath.

For example, the following statement:

```
bmj -d jbproject/project1/classes tester.java
```

causes the class files for the classes defined in the `tester.java` source file to be saved in the directory `jbproject/project1/classes/test/test3`, assuming that `tester.java` contains the following package statement:

```
package test.test3;
```

The updated dependency file, `test.test3.dep2`, is saved in `jbproject/project1/classes/package cache`.

Files are read from the class path and written to the destination directory. The destination directory must be part of the class path. The default destination for class files matches the package structure in the source files and starts from the root directory of the source. The default destination for dependency files matches the package structure and starts in the current directory.

- **-deprecation**

Displays all deprecated classes, methods, properties, events, and variables used in the API.

If a warning is displayed when compiling, indicating that some deprecated APIs were used, you can turn this option on to see all deprecated APIs.

- **-encoding** *name*

You can specify a file-encoding name (or `codepage` name) to control how the compiler interprets characters beyond the ASCII character set. The

default is to use the default native-encoding converter for the platform. For more information, see [“Specifying a native encoding for the compiler” on page 16-12.](#)

For example, the following statement:

```
bmj -encoding EUC_JP tester.java
```

compiles `tester.java` and any directly imported `.java` files that do not have `.class` files. All source files are interpreted as being encoded in the EUC\_JP character set, which is the character set typically used for Japanese UNIX environments. You can specify any encoding that is supported by the Java 2 platform. A list of the valid encodings is available at <http://java.sun.com/j2se/1.4/docs/tooldocs/tools.html#intl>.

- **-exclude** `classname`

Excludes all calls to `static void` methods in the selected `.class` file from a compile. This also excludes the evaluation of the parameters passed to those methods.

For example, excluding class A removes all calls to `static void` methods of A from OTHER classes.

- **-g**

Generates all debugging information in the class file, including local variables. By default, only line numbers and source file information is generated.

- **-g:none**

Doesn't generate any debugging information.

- **-g:{keyword list}**

Generates some types of debugging information. The keyword list is a comma-separated list of keywords, for example: `bmj -g:source,lines`

Keywords include:

- **source:** source file debugging information.
- **lines:** line number debugging information.
- **vars:** local variable debugging information.

- **-nocompile**

Checks whether the classes are up-to-date, but doesn't compile any classes. Useful for quickly checking whether a class or package is up to date. Stops at the first file needing recompilation, and reports "Class <class> needs recompiling because <reason>."

- **-nowarn**

Compiles without displaying warnings.

- **-obfuscate**

Obfuscation makes your programs less vulnerable to reverse engineering. After decompiling your obfuscated code, the generated source code contains altered symbol names for private symbols.

- **-quiet**

Compiles without displaying any messages.

- **-rebuild**

The Rebuild command compiles the specified root classes and their imported files, regardless of whether they have changed.

### See also

- [“The Rebuild command” on page 6-4](#)

- **-source** version

Enables support for compiling source code containing assertions.

- When the version is set to 1.4, this command-line option enables the `assert` keyword and the JDK 1.4 assertion facility.
- When the version is set to 1.3, the compiler does not support assertions.
- If the `-source` option isn’t used, the compiler defaults to the 1.3 behavior.

### See also

- “Assertion Facility” at <http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>

- **-sourcepath** path

The path used to find sources. If no source path is specified, the classpath is used to find the sources.

Similar to the classpath, the sourcepath must point to the root of the package directory tree, and not directly to the directory of the sources.

For example, to make `tester.java`, which contains the package statement `package test.test3;` and is located in `jbproject/project1/src/test/test3`, you must set the source path to `jbproject/project1/src` and not to `jbproject/project1/src/test/test3`.

You can then type the following:

```
bmj -sourcepath jbproject/project1/src
jbproject/project1/src/test/test3/tester.java
-d jbproject/project1/classes
```

- **-sync**

Deletes class files on the outpath that you don't have source files for before compiling. You have to specify the outpath directory using the **-d** option.

This option can be helpful to avoid situations when the compiler might find a class file in the output directory, which cannot be compiled from source. (You may have deleted the source file or you have renamed one of the classes declared in a source file.)

- **-verbose**

This option gives more information about compiling, such as which class files are loaded from where in the classpath. The following information is displayed:

- The classpath, sourcepath, and output directory that are being used.
- Which source files are being compiled.
- Which classes files are loaded.
- Which classes are generated.
- Which dependency files are generated.

## Cross-compilation options

---

**bmj** supports cross-compilation, where classes are compiled against a bootstrap and extension classes of a different Java platform. Use **-bootclasspath** and **-extdirs** when cross-compiling.

- **-target** *version*

Restricts the class files to work on a specific VM version.

**bmj** supports:

- 1.1 - Generates class files to run on 1.1 and VMs in the Java 2 SDK. When you select this as the target VM, your class files can be loaded by any VM.
- 1.2 - Generates class files to run **only** on VMs in the Java 2 SDK, v 1.2 and later, but **won't** run on 1.1 VMs. This is the default.
- 1.3 - Generates class files to run **only** on VMs in the Java 2 SDK, v 1.3 and later, but **won't** run on 1.1 or 1.2 VMs.
- 1.4 - Generates class files to run **only** on VMs in the Java 2 SDK, v 1.4 and later, but **won't** run on 1.1, 1.2, or 1.3 VMs.
- **-bootclasspath** *bootclasspath*

Cross-compile against the specified boot classes. Entries can be directories, JAR archives, or ZIP archives.

- **-extdirs** directories

Cross-compile against the specified extension directories. Each JAR file in the specified directories is searched for class files.

## Specifiers for root classes

---

Root classes are specified in the following form:

```
{[-s] {source.java} | -p {package} | -c {class}}
```

- **-s** sourcefilename

Indicates that the specified root classes are those defined in the given source files. This is the default interpretation.

For example, the following statement:

```
bmj -sourcepath jbpjproject/project1/src
-s jbpjproject/project1/src/tester.java
```

is the same as

```
bmj -sourcepath jbpjproject/project1/src
jbpjproject/project1/src/tester.java
```

If you list some packages with the **-p** option before listing sources, then you need to specify the **-s** option. If you list sources before packages and classes, the **-s** is assumed and is not necessary.

- **-p** packagename

The name of packages to compile.

For example, the following statement:

```
bmj -sourcepath jbpjproject/project1/src -p test.test3
```

makes all classes of the `test.test3` package and all imported classes.

- **-c** classname

The names of the classes to make.

For example, the following statement:

```
bmj -sourcepath jbpjproject/project1/src
-c test.test3.testter
```

makes the class `testter` of package `test.test3` and all imported classes.

As another example, the following statement:

```
bmj -sourcepath jbpjproject/project1/src tester.java -p package1 package2
-s jbpjproject/project1/src/*.java
```

makes the source file `tester.java`, packages `package1` and `package2`, and all the java files in the `jbpjproject/project1/src` directory.

Note that the first source name (`tester.java`) comes before the **-p** (package) option so does not need to explicitly specify the **-s** option, because **-s** is assumed. If you want to specify another source file name after the **-p** option is specified, you have to explicitly specify the **-s** option.

## VM options

---

- **-Joption**

Passes options to the `java` launcher called by **bmj**. For example, **-J-Xms48m** sets the startup memory to 48 megabytes. It is a common convention for **-J** to pass options to the underlying VM executing applications written in Java.

Note that `CLASSPATH`, **-classpath**, **-bootclasspath**, and **-extdirs** do not specify the classes used to run **bmj**. If you do need to do this, use the **-J** option to pass through options to the **bmj** launcher.



# Index

## Symbols

---

- @author 14-5
- @deprecated 14-5
- @docRoot 14-5
- @exception 14-5
- @link 14-5
- @param 14-5
- @return 14-5
- @see 14-5
- @serial 14-5
- @serialData 14-5
- @serialField 14-5
- @since 14-5
- @tags
  - list of 14-5
- @throws 14-5
- @version 14-5

## A

---

- adding a JDK 2-20
- adding directory view to project 2-16
- adding events
  - to JavaBeans 10-11, 10-14, 10-15
- adding libraries 4-2
- adding to projects 2-13
  - See also* projects
- Ant 6-7
  - adding targets to Project menu 6-18
  - Ant wizard 6-7
  - build files 6-7
  - building 6-7
  - importing Ant projects 6-7
  - libraries 6-14
  - options 6-14
  - setting JDK 6-12
  - setting properties 6-12
  - targets 6-7
  - tutorial 18-1
  - using bmv command-line make 6-12
- Ant build files
  - adding to project 6-7
- Ant wizard 6-7
- API documentation
  - creating 14-1
  - generating output files 14-16
  - viewing from project pane 14-20
  - viewing in Doc tab 14-20, 14-22
  - viewing in Help Viewer 14-20
  - API documentation viewing
    - in UML browser 11-13
- applets
  - deploying 15-1, 15-11
  - JDK versions 15-8
  - running 7-1
- appletviewer
  - command-line tool B-1
- applications
  - building 6-1
  - compiling 5-1
  - debugging 8-1
  - deploying 15-1, 15-11
  - running 7-1
  - testing 13-1
- Archive Builder
  - archive types 15-17
  - creating archive files 15-4
  - creating documentation archive 14-25
  - creating executable files 15-26
  - deploying files 15-17
  - setting runtime configuration options 15-28
- archive files
  - creating executable files 15-26
  - creating for deployment 15-2
  - creating for documentation 14-25
  - deleting 15-33
  - JAR files 15-4
  - manifest files 15-3
  - removing 15-33
  - renaming 15-33
  - setting runtime configuration options 15-28
  - types supported in Archive Builder 15-17
  - viewing contents 15-6, 15-31
  - ZIP files 15-4
- archive node
  - building 15-31
  - deleting 15-33
  - properties 15-32
  - removing 15-33
  - renaming 15-33
  - resetting properties 15-31
  - viewing contents 15-31
- archiving
  - documentation 15-17
  - manifest file 15-3
  - projects for deployment 15-2
  - source files 15-17
  - with Archive Builder 15-4

- with jar tool 15-5
- with Native Executable Builder 15-29
- Assert class 13-6
- assert keyword
  - enabling 2-4, B-10, B-15
- assertEquals() 13-6
- assertNotNull() 13-6
- assertTrue() 13-6
- auto source package settings 2-4
- automatic source packages
  - Project Source node 6-21
- automatic source packages option
  - deepest package exposed 6-21

## B

- backup path 4-12
- bcj command-line compiler 5-9, B-7
- BeanInfo classes
  - creating 10-8
  - generating automatically 10-9
  - modifying 10-10
- BeanInfo data 10-9
  - modifying 10-9
- BeanInfo designer 10-9
- BeanInsight 10-21
- BeansExpress 10-1
  - changing BeanInfo classes 10-10
  - changing properties 10-6
  - creating JavaBeans 10-2
  - removing properties 10-7
  - setting properties 10-4, 10-7
- bmj command-line make 5-9, B-12
  - using with Ant 6-12
- Borland
  - contacting 1-6
  - developer support 1-6
  - e-mail 1-7
  - newsgroups 1-7
  - online resources 1-6
  - reporting bugs 1-7
  - technical support 1-6
  - World Wide Web 1-6
- Borland Compiler for Java (bcj) B-7
- Borland Make for Java (bmj) B-12
- bound properties
  - setting for JavaBeans 10-7
- breakpoints 8-40
  - actions for 8-51
  - class 8-45
  - conditional 8-52, 8-53
  - cross-process 8-47, 9-10
  - disabling 8-54
  - enabling 8-54
  - exception 8-43
  - field 8-47
  - for debugging tests 13-17
  - line 8-41
  - locating 8-55
  - method 8-46
  - overriding Tracing Disabled 8-40
  - pass counts 8-53
  - properties 8-50
  - removing 8-54
  - running to 8-35
  - setting 8-41
- browse path 4-11
- build command-line option 5-10
- build files
  - adding to project 6-7
- build menus 5-3
  - adding targets 6-18, 6-19
- build phases
  - Clean 6-2
  - Compile 6-2
  - Deploy 6-2
  - Make 6-2
  - Package 6-2
  - Post-compile 6-2
  - Pre-compile 6-2
  - Rebuild 6-2
- build target
  - runtime configuration 7-10
- build targets 5-4
  - Ant 6-7
- build tasks 6-16
  - building external tasks 6-17
  - External Build Task wizard 6-16
  - setting Ant properties 6-12
  - setting properties 6-17
- building
  - See also* compiling
  - Ant 6-7
  - Ant messages 6-11
  - automatic source packages 6-21
  - build messages 6-17
  - build tasks 6-2
  - Clean command 6-5
  - copying resources to output path 6-25
  - creating external build tasks 6-16
  - excluding packages 6-23
  - external build tasks 6-16
  - from command line B-4
  - Java programs 6-1
  - Make command 6-3
  - overview 6-1
  - phases 6-2
  - project groups 6-5

- Rebuild command 6-4
- Run command 5-4
- Run command and Ant 6-12
- SQLJ files 6-15
- targets 6-2
- terms defined 6-2

## C

- Cactus 13-2, 13-11
  - configuring your project 13-11
  - running tests 13-13
  - testing an EJB 13-7
- Cactus Setup wizard 13-11
- calls to methods
  - viewing 8-36
- changing
  - BeanInfo classes 10-10
  - data values in debugger 8-57
  - method parameters 12-22
- class breakpoints 8-45
- class files 4-10
  - directory locations for 4-8
  - how JBuilder finds 4-13
- class path 4-10
- classes
  - API documentation for 14-2
  - finding definition of 12-7
  - finding references to 12-8
  - redistribution in deployment 15-15
  - UML diagrams 11-4, 11-7
  - updating after compiling 8-64
  - updating after compiling (options) 8-65
- Classes with tracing disabled view 8-12
- classic option for debugging 8-7
- classpath
  - setting paths B-2
- CLASSPATH environment variable
  - setting for command line B-2
- Clean command 6-5
- Clean phase 6-2
- code
  - modifying while debugging 8-64
  - obfuscated code in UML diagrams 11-3
  - viewing from a UML diagram 11-13
  - visualizing
    - See also* UML
- code block
  - surrounding with try/catch 12-26
- code symbols
  - discovery before refactoring 12-7
  - refactoring 12-1
- codepages 16-14
  - See also* native encodings
- command line
  - building projects 5-10
  - compiling 5-9
  - JBuilder arguments 5-10
  - JBuilder command-line interface B-4
  - running deployed programs 7-13
  - running JAR files 7-13
  - switching to IDE when compiling 5-11
  - tools B-1
    - See also* command-line tools
- command-line compilers
  - bcj 5-9, B-7
  - bmj 5-9, B-12
- command-line interface
  - JBuilder B-4
- command-line options
  - for Javadoc 14-13
  - in Javadoc wizard 14-13
- command-line tools B-1
  - appletviewer B-1
  - native2ascii B-1
  - bcj B-7
  - bmj B-12
  - jar 15-5, B-1
  - java B-1
  - javac B-1
  - javadoc B-1
  - JBuilder B-4
  - setting CLASSPATH B-2
  - setting paths for B-2
- comments for Javadoc 14-2
- comparison fixture 13-10
  - tutorial 22-1
  - wizard 13-10
- Compile phase 6-2
- compilers
  - bcj 5-9, B-7
  - bmj 5-9, B-12
  - Borland Make 5-8
  - errors 5-4
  - javac 5-8
  - project javac 5-8
  - setting options 5-6
  - specifying in IDE 5-8
- compiling
  - See also* building
  - before refactoring 12-6
  - building with Ant files 18-1
  - Clean command 6-5
  - copying resources to output path 6-25
  - errors 5-4
  - errors when opening projects 5-5
  - excluding packages 6-23
  - from command line 5-9

- how JBuilder finds files 4-13
- Java programs 5-1
- Make command 6-3
- overview 5-1
- programs with debug info 8-4
- project groups 6-5
- projects within a project group 5-9
- project-wide settings 5-6
- Rebuild command 6-4
- Run command 5-4
- setting options 5-6
- setting output path 5-8
- setting target VM 5-6
- smart dependencies checking 5-2
- source files 5-3
- specifying a different compiler 5-8
- status bar 7-2
- tutorial 17-1
- with references from project libraries 12-6
- compiling in IDE
  - with Borland Make for Java 5-8
  - with javac 5-8
  - with project javac 5-8
- component palette
  - installing JavaBeans 10-22
- components
  - cross-platform 10-1
  - reusable 10-1
- conditional breakpoints 8-52
  - setting 8-53
- configuration
  - runtime types 7-12
- configuration files A-1
- configurations
  - displaying with command-line argument B-4
- configuring
  - JDKs in JBuilder 2-22
  - libraries 4-2
- Console output, input, and errors view 8-11
- constrained properties
  - setting for JavaBeans 10-7
- creating
  - API documentation 14-1
  - BeanInfo classes 10-8
  - custom doclet 14-27
  - JavaBeans 10-1, 10-2
  - property editors 10-17
  - runtime configurations 7-6, 7-8
  - test cases 13-5
- creating project files 2-11
- creating projects from existing files 2-7
- cross-platform components 10-1
- cross-process breakpoints 8-47, 9-10

- cursor location
  - running to 8-36
- custom fixture 13-11
  - wizard 13-11
- customizing
  - JavaBeans 10-8

## D

---

- Data and code breakpoints view 8-13
- data values
  - changing in debugger 8-57
  - examining while debugging 8-55
- Data watches view 8-18
- deadlocks 8-32
- debug information 8-4
- debugger 8-1
  - and unit tests 13-17
  - customizing display 8-67
  - customizing settings 8-68
  - ending session 8-8
  - ExpressionInsight 8-25
  - pausing program execution 8-8
  - running under control of 8-7
  - setting update intervals 8-69
  - shortcut keys 8-25
  - status bar 8-23
  - tool tips 8-26
  - toolbar 8-23
  - user interface 8-8
  - views 8-10
- debugger views
  - Classes with tracing disabled view 8-12
  - Console output, input, and errors view 8-11
  - Data and code breakpoints view 8-13
  - Data watches view 8-18
  - Loaded classes and static data view 8-20
  - Synchronization monitors view 8-22
  - Threads, call stacks, and data view 8-15
- debugging 8-1
  - attaching to running program 9-6
  - breakpoints and tracing disabled settings 8-40
  - changing data values 8-57
  - choosing stepping thread 8-31
  - class file 8-6
  - compiling with debug info 8-4
  - controlling program execution 8-27
  - controlling tracing into classes 8-37
  - cross-process breakpoints 9-10
  - deleting watches 8-62
  - detecting deadlocks 8-32
  - displaying current thread 8-31
  - displaying static and local variables 8-56
  - displaying top stack frame 8-31

- distributed applications 9-1
- editing watches 8-61
- examining program data values 8-55
- execution point 8-28
- from command line 8-1
- JSP 8-27
- keeping thread suspended 8-31
- launching program remotely 9-2
- LegacyJ code 8-27
- local code running in separate process 9-9
- logic errors 8-2
- managing threads 8-30
- modifying code 8-64
- modifying values 8-63
- moving through code 8-32
- non-Java source 8-27
- overview 8-3
- project 8-6
- remotely 9-1
- resetting program 8-28
- running program on remote computer 9-6
- running to a breakpoint 8-35
- running to the cursor location 8-36
- running to the end of a method 8-36
- runtime configuration 8-3
- runtime exceptions 8-2
- sessions 8-9
- setting breakpoints 8-40, 8-41
- setting execution point 8-29
- smart step 8-34
- SQLJ code 8-27
- starting session 8-6
- starting session with -classic 8-7
- stepping into method calls 8-33
- stepping out of a method 8-34
- suspending and running debugger 8-27
  - tracing into classes with no source available 8-39
- tutorial 17-1
- unit test 8-6
- unit tests 13-3, 13-17
- using -classic option 2-20
- viewing method calls 8-36
- watching expressions 8-59
- web application 8-6
- deleting 2-15
- dependencies
  - UML diagrams 11-7
- dependencies checking 5-2, 5-6
- Deploy phase 6-2
- deploying
  - applets 15-11
  - applications 15-1, 15-11
  - applications as archives 15-2
  - distributed applications 15-15
  - JavaBeans 15-13
  - tips 15-14
  - to Internet 15-14
- deployment issues 15-7
  - applets relying on JDK 1.1.x or Java 2 15-8
  - applets vs. applications 15-9
  - download time 15-10
  - libraries 15-9
  - libraries on CLASSPATH 15-8
  - redistribution of classes 15-15
- design
  - patterns (JavaBean) 10-1
- designing
  - JavaBean user interfaces 10-4
- diagrams, UML 11-10
  - See also* UML
- directory view
  - adding 2-16
- directory-package correspondence 5-6
- display of static and local variables 8-56
- displaying
  - project files 4-13
- distributed applications
  - debugging 9-1
  - deploying 15-15
- doc path 4-11
- doclet
  - creating custom 14-27
  - JDK 1.1 14-9
  - Standard 14-9
- documentation
  - viewing Javadoc 14-22
- documentation archive
  - creating 14-25
- documentation conventions 1-4
  - platform conventions 1-5
- documentation node
  - changing properties for 14-23, 14-24
  - expanding 14-20
  - generating 14-8
  - Javadoc wizard 14-8
  - properties for 14-8
- duplicate class definitions 5-6

## E

---

- editing the JDK 2-19
- editor
  - and test runners 13-14
  - optimizing imports 12-14
- EJB
  - testing 13-11
  - testing with Cactus 13-12

- EJB Test Client wizard 13-7, 13-12
- encoding
  - setting 2-4
- encodings
  - adding and overriding 16-13
  - native 16-12
  - native defined 16-2
  - setting options 16-12
- error messages 5-4
  - debugger 8-2
- errors
  - error messages 5-4
  - logic 8-2
  - runtime 8-2
  - syntax errors 5-4
  - types of 8-2
- evaluating and modifying expressions 8-62
- event sets
  - creating custom 10-15
- events
  - adding to JavaBeans 10-11, 10-14, 10-15
- exception breakpoints 8-43
- exceptions
  - and unit tests 13-6
- executable files
  - creating 15-26
  - running 15-26
- executables
  - configuration files A-1
  - launcher A-1
- execution point
  - overview 8-28
  - setting 8-29
- Expose Superclass BeanInfo option 10-9
- ExpressionInsight 8-25
- expressions
  - evaluating 8-62
  - evaluating and modifying 8-62
  - watching 8-59
- External Build Task wizard 6-16
- external build tasks 6-16
- extract method 12-4
- extracting
  - method 12-24

## F

---

- field API documentation 14-2
- field breakpoints 8-47
- fields
  - finding definition of 12-7
  - finding references to 12-8
  - UML diagrams 11-7

- files
  - generated for Javadoc 14-16
  - location 4-12
    - See also* project files
  - opening outside of project 2-15
  - renaming 2-15
  - running within project 7-1
  - stub source 8-39
  - switching 2-9
  - viewing in JBuilder 2-9
  - viewing project files 4-13
- Files Modified dialog box 8-65
- filtering
  - excluding packages from build 6-23
- filtering packages 6-23
- Find Definition command 12-7
- Find References 12-8
  - in the UML browser 11-21
- finding definition of
  - class 12-7
  - field 12-7
  - method 12-7
  - variable 12-7
- finding references to
  - class 12-8
  - field 12-8
  - method 12-8
  - variable 12-8
- fonts
  - displaying international fonts 16-10
  - JBuilder documentation conventions 1-4

## G

---

- generating
  - JavaBeans 10-2
  - Javadoc 14-1
  - Javadoc output files 14-16
  - Javadoc tags 14-6
- grouped projects
  - running 7-5

## I

---

- icons
  - JBTestRunner 13-14
  - specifying JavaBean 10-9
  - UML visibility icons 11-9
- IDE Options
  - UML 11-19
- image files
  - saving UML diagrams 11-20
- import statements 4-8
  - optimizing 12-14

- imports
  - optimizing 12-2
- installing
  - JavaBeans on component palette 10-22
- interface API documentation 14-2
- interfaces
  - UML diagrams 11-7
- international versions
  - JBuilder 16-15
- internationalization 16-2, 16-16
  - compiler features 16-12
  - dbSwing 16-8
  - debugger features 16-12
  - defined 16-1
  - encoding options 16-12
  - JBuilder features 16-2
  - non-native peers 16-10
  - programs 16-1
  - terms and definitions 16-1
  - UI designer features 16-10
- Internet
  - deploying applications to 15-14
- introduce variable 12-4

## J

---

- jar command-line tool B-1
- JAR files 15-4
  - adding to project 4-1
  - command-line access 15-5
  - creating with Archive Builder 15-4
  - creating with jar tool 15-5
  - extracting 15-6
  - manifest 15-3
  - running from command line 7-13, 15-24
  - updating from command line 15-7
  - viewing contents 15-6
- Java
  - and UML 11-2
- Java archive files 15-4
  - See also* JAR files
- Java Archive Tool 15-5
- Java classes
  - collections of 10-1
- java command-line tool B-1
- Java files
  - directory locations for 4-7
- Java initialization string 10-18
- JavaBean wizard 10-2
- JavaBeans 10-2
  - adding events 10-11, 10-14, 10-15
  - advantages 10-1
  - changing properties for 10-6
  - creating 10-1, 10-2
  - customizing 10-8
  - defined 10-1
  - deploying 15-13
  - designing user interface 10-4
  - displaying property settings 10-9
  - installing 10-22
  - removing properties for 10-7
  - serializing 10-21
  - setting properties for 10-4, 10-7
  - validity 10-21
- javac command-line tool B-1
- javadoc command-line tool B-1
- Javadoc comments
  - adding for classes 14-3
  - adding for fields 14-3
  - adding for interfaces 14-3
  - adding for methods 14-3
  - automatically generating tags for 14-6
  - conflicts 14-8
  - examples of 14-2
  - todo tags 14-7
  - using tags 14-5
  - where to place 14-3
- Javadoc documentation 14-1
  - adding comments 14-2
  - build options 14-10
  - command-line options 14-13
  - formatting output 14-9
  - generated files 14-16
  - generating output files 14-16
  - generating package files 14-18
  - JDK 1.1 output 14-9
  - maintaining 14-22
  - name of node 14-10
  - output directory 14-10
  - overview comment files 14-18
  - package detail files 14-18
  - scope of documentation 14-12
  - Standard output 14-9
  - viewing for entire project 14-20
  - viewing for individual file 14-20, 14-22
  - viewing from project pane 14-20
  - viewing from UML diagram 14-20
  - viewing in Doc tab 14-20, 14-22
  - viewing in Help Viewer 14-20
- Javadoc documentation viewing
  - from UML diagram 11-13
- Javadoc fields
  - setting in Project wizard 2-4
- Javadoc tags
  - @author 14-5
  - @deprecated 14-5
  - @docRoot 14-5
  - @exception 14-5

- @link 14-5
- @param 14-5
- @return 14-5
- @see 14-5
- @serial 14-5
- @serialData 14-5
- @serialField 14-5
- @since 14-5
- @throws 14-5
- @version 14-5
- automatically generating 14-6
- entering in source files 14-5
- list of 14-5
- Javadoc wizard 14-1
  - build options 14-10
  - changing properties for node 14-23
  - command-line options 14-13
  - formatting output 14-9
  - JDK 1.1 output 14-9
  - name of documentation node 14-10
  - output directory 14-10
  - scope of documentation 14-12
  - Standard output 14-9
- JBTestRunner 13-14
  - and editor 13-14
  - icons 13-14
  - Test Failures 13-14
  - Test Hierarchy 13-14
  - Test Output 13-14
  - tutorial 21-1
- JBuilder
  - international versions 16-15
- JDBC fixture 13-8
  - tutorial 22-1
  - wizard 13-8
- JDK
  - setting for Ant projects 6-12
- JDK 1.1 output for Javadoc 14-9
- JDKs
  - configuring in JBuilder 2-22
  - debugging with -classic 2-20
  - editing 2-19
  - setting in the Project wizard 2-4
  - switching and setting 2-20
- JNDI fixture 13-9
  - wizard 13-9
- JSP
  - debugging 8-27
- JUnit 13-1
  - Assert class 13-6
  - assertEquals() 13-6
  - assertNotNull() 13-6
  - assertTrue() 13-6
  - AwtUI 13-1

- integration into JBuilder 13-1
- setUp() 13-4
- SwingUI 13-1, 13-16
- tearDown() 13-4
- test runners 13-1
- TestCase class 13-1, 13-3
- testing an EJB 13-7
- TestSuite class 13-1, 13-3
- TextUI 13-1, 13-16
- JUnit Test Collector 13-3

## L

---

- launcher
  - configuration files A-1
  - executables A-1
- LegacyJ code
  - debugging 8-27
- libraries
  - adding and configuring 4-2
  - adding projects as required 3-4, 4-5
  - adding to project 2-4
  - Ant 6-14
  - defined 4-1
  - deployment 15-9
  - display lists of 4-6
  - editing 4-5
  - including references in UML diagrams 11-18, 11-19
  - redistributable 15-9
  - setting paths 2-22
- line breakpoints 8-41
- Loaded classes and static data view 8-20
- local variable display 8-56
- locale
  - defined 16-2
- locale-sensitive features 16-9
- Localizable Property Setting dialog box 16-8
- localization 16-1
  - defined 16-2
- locating
  - a method call 8-37
- logic errors 8-2

## M

---

- main class
  - setting 7-3
- maintaining Javadoc 14-22
- make
  - command-line bmj B-12
- Make command 6-3
- Make phase 6-2



- manifest files 15-3
  - editing 15-24
  - viewing in JBuilder 15-31
- menus
  - configuring Project Group menu 6-19
  - configuring Project menu 6-18
- method API documentation 14-2
- method breakpoints 8-46
- method calls
  - evaluating 8-62
  - locating 8-37
  - viewing 8-36
- method parameters
  - changing 12-4
- methods
  - finding definition of 12-7
  - finding references to 12-8
  - running to end of 8-36
  - UML diagrams 11-7
- modifying
  - code while debugging 8-64
  - values of variables 8-63
- move refactoring
  - classes 12-3, 12-18
  - compiling before 12-6
  - overview 12-1
  - previewing 12-10
  - saving 12-26
  - symbol discovery 12-7
  - undoing 12-26
- moving
  - classes 12-3, 12-18
  - saving 12-26
  - undoing 12-26
- multilingual sample application 16-3
- multiple projects 2-23
  - saving 2-24
  - switching between 2-23

## N

---

- naming conventions
  - packages 4-9
- native encodings 16-12, 16-14
  - defined 16-2
  - supported 16-13
- Native Executable Builder 15-29
- native executable node
  - setting properties 15-30
- native2ascii command-line tool B-1
- navigating
  - UML diagrams 11-15
- New Directory View command 2-16
- New Property Editor dialog box 10-17

- newsgroups
  - Borland 1-7
  - public 1-7
- nodes
  - Project Source 6-21
- non-Java source
  - debugging 8-27

## O

---

- obfuscated code
  - in UML diagrams 11-3
- object watches 8-61
- opening files
  - outside of project 2-15
- opening project files 2-10
- OpenTools
  - enabling verbose debugging mode with command-line argument B-4
  - running 7-5
- Optimize Imports command 12-14
- optimizing imports 12-2
- organizing projects 2-13
  - See also* projects
- out path 4-10
- output path
  - setting 5-8

## P

---

- package discovery 6-21
- package file
  - Javadoc documentation 14-18
- Package Filters folder 6-23
- Package phase 6-2
- package-directory correspondence 5-6
- packages 4-6
  - automatic source packages 6-21
  - class reference in 4-8
  - enable source package discovery 6-21
  - excluding from build process 6-23
  - filtering 6-23
  - naming conventions 4-9
  - source package discovery 6-21
  - UML diagrams 11-4, 11-7
- PackageTestSuite class 13-3
- Palette Properties dialog box 10-22
- pass count breakpoints 8-53
- paths 4-1
  - See also* setting paths
  - class file locations 4-8
  - compiling, running, debugging 4-13
  - Java file locations 4-7
  - setting B-2

- setting output path 5-8
  - setting with classpath option B-2
- pausing program execution 8-8
- Post-compile phase 6-2
- Pre-compile phase 6-2
- program data values 8-55
- program execution
  - controlling 8-27
- programs
  - building 6-1
  - compiling 5-1
  - debugging 8-1
  - deploying 15-1
  - running 8-27
  - suspending 8-27
  - testing 13-1
- project
  - configuring for Cactus 13-11
  - setting general parameters 2-4
- project files 2-1
  - paths 4-12
    - See also* setting paths
  - saving 2-10
  - setting paths 4-9
  - viewing 4-13
- Project For Existing Code wizard 2-7
- project group file 3-1
- Project Group menu
  - adding targets 6-19
- Project Group wizard 3-1
- project groups 3-1
  - adding projects 3-1, 3-3
  - adding targets to Project menu 6-19
  - advantages 3-1
  - build order of projects 3-1
  - building 6-5
  - compiling 6-5
  - configuring Project Group menu 6-19
  - creating 3-1
  - navigating in 3-4
  - removing projects 3-3
  - running 7-5
- Project menu
  - adding targets 6-18
  - configuring 6-18
  - configuring for project groups 6-19
  - Make command 6-18
  - Rebuild command 6-18
- project paths
  - setting 2-4
- project properties
  - setting for refactoring 12-6
  - setting for references discovery 12-6
  - setting for UML 11-17
- setting output path 5-8
  - source directory for tests 13-5
- Project Source node 6-21, 6-23
  - filtering packages 6-23
- project template
  - setting 2-2
- Project wizard
  - creating new projects 2-2
  - Step 1-setting root, type, and template 2-2
  - Step 2-setting paths, libraries,JDK 2-4
  - Step 3-general project settings 2-4
- projects 2-1
  - adding as required libraries 3-4, 4-5
  - adding directory new 2-16
  - adding files or packages 2-13
  - adding files, packages 2-14
  - adding folders 2-13
  - adding to project groups 3-1
  - adding ZIP or JAR files 4-1
  - Ant 6-7
  - building 6-1
  - building Ant with Run command 6-12
  - building from command line 5-10, B-4
  - building with Run command 5-4
  - closing 2-10
  - compiling 5-1
  - configuring Project menu 6-18
  - creating and adding to 2-11
  - creating new 2-2
  - creating with Project For Existing Code wizard 2-7
  - creating with Project wizard *See* Project wizard
  - opening 2-10
  - removing from 2-14
  - renaming 2-15
  - running 7-1, 7-3
  - saving 2-10
  - saving multiple 2-24
  - setting main class 7-3
  - setting properties 2-17
  - switching between 2-23
  - using multiple projects 2-23
  - viewing files 2-9
- projects groups
  - project dependencies 3-4
- properties
  - Ant 6-12
  - archive node 15-31
  - changing 10-17
  - customizing 10-9
  - external build task 6-17
  - hiding 10-9
  - JavaBean components 10-4, 10-7

- Native Executable Builder 15-30
- UML diagrams 11-7
- property editors 10-9
  - creating 10-17
  - custom component 10-20
  - Java initialization string 10-18
  - String List 10-17
  - String Tag List 10-18
- PropertyChangeSupport class 10-7

## R

---

- readObject() 10-21
- Rebuild command 6-4
- Rebuild phase 6-2
- redistribution of classes 15-15
- refactoring
  - change method parameters 12-4
  - changing method parameters 12-22
  - extract method 12-4, 12-24
  - from the UML browser 11-21
  - introduce variable 12-4, 12-25
  - move 12-1
    - See also* move refactoring
  - optimize imports 12-2
  - rename 12-1
    - See also* rename refactoring
  - surround block with try/catch 12-26
  - surround with try/catch 12-4
- reference types
  - ancestors 12-8
  - declarations 12-8
  - descendents 12-8
  - descendents member references 12-8
  - descendents type references 12-8
  - direct usages 12-8
  - indirect usages 12-8
  - member references 12-8
  - reads 12-8
  - type references 12-8
  - writes 12-8
- references
  - from libraries 2-4
  - setting up for discovery of 12-6
- referencing classes 4-8
- regression testing 13-1
  - See also* unit testing
- remote debugging 9-1
- remote debugging tutorial 19-1
- Remove Watch command 8-62
- rename refactoring
  - classes 12-2, 12-17
  - compiling before 12-6
  - fields 12-2, 12-21
  - local variables 12-2, 12-20
  - methods 12-2, 12-19
  - overview 12-1
  - packages 12-2, 12-17
  - previewing 12-10
  - properties 12-2, 12-22
  - saving 12-26
  - symbol discovery 12-7
  - undoing 12-26
- renaming
  - classes 12-2, 12-17
  - fields 12-2, 12-21
  - local variables 12-2, 12-20
  - methods 12-2, 12-19
  - packages 12-2, 12-17
  - properties 12-2, 12-22
  - saving 12-26
  - undoing 12-26
- required libraries
  - adding projects 3-4, 4-5
- Resource Strings wizard 16-5
- resources
  - automatic source packages 6-21
  - copying to output path 6-25
- resourcing
  - defined 16-2
  - strings 16-5
- reusable components 10-1
- Run command 7-3
  - building 5-4
  - building Ant 6-12
- Run To Cursor command 8-36
- Run To End Of Method command 8-36
- runnable class
  - setting for project 7-3
  - specifying in manifest file 15-24
- runner types 7-12
- running
  - applets 7-1
  - applications 7-1
  - applications from command line 7-13
  - Cactus tests 13-13
  - grouped projects 7-5
  - individual files 7-1
  - OpenTools 7-5
  - projects 7-3
  - tutorial 17-1
  - under debugger control 8-7
  - unit tests 13-3, 13-14
  - web files 7-2
- runtime
  - errors 8-2
  - exceptions 8-2

- runtime configuration options
  - setting in archive file 15-28
- runtime configuration types 7-12
- runtime configurations
  - build target 7-10
  - creating 7-8
  - debugging 8-3
  - for unit testing 13-16
  - setting 7-6
  - Test 13-3

## S

---

- samples
  - multilingual application 16-3
- saving multiple projects 2-24
- separate process debugging 9-9
- serialization
  - adding support for 10-21
- server-side testing 13-2, 13-11
- Set Execution Point 8-29
- setting JDKs 2-20
- setting paths 4-9
  - backup 4-12
  - browse path 4-11
  - class path 4-10
  - CLASSPATH B-2
  - classpath option B-2
  - command-line tools B-2
  - doc 4-11
  - JDKs 2-19
    - See also* JDKs
  - output 4-10
  - project paths 2-4
  - required libraries 2-22
  - source files 4-10
- setting properties
  - in JavaBeans 10-4, 10-7
  - projects 2-17
- setting runtime configurations 7-6
- setUp() 13-4
- smart dependencies checking 5-2
- Smart Source 8-27
- Smart Step 8-33, 8-34
  - options 8-34
- Smart Swap 8-64
  - and the target VM 8-64
- source directory
  - test 13-5
- source discovery 6-21
- source files
  - path 4-10

- source paths
  - class files 4-8
  - Java files 4-7
- splash screen
  - disabling with command-line argument B-4
- SQLJ
  - building files 6-15
- SQLJ code
  - debugging 8-27
- stack trace filter
  - unit testing 13-16
- Standard output for Javadoc 14-9
- static variable display 8-56
- status bars
  - build progress 7-2
  - debugger 8-23
- Step Into 8-33
- step into 8-6
- Step Out 8-34
- Step Over 8-33
- step over 8-6
- stepping
  - into method calls 8-33
  - out of a method 8-34
  - over method calls 8-33
  - smart 8-33, 8-34
- structure pane
  - Errors folder 5-4
  - Javadoc conflicts 14-8
  - ToDo folder 14-7
  - UML 11-16
- stubs
  - source files 8-39
- superclasses 10-9
- Support Serialization option 10-21
- surround with try/catch 12-4
- SwingUI 13-16
- switching files 2-9
- switching the JDK 2-20
- Synchronization monitors view 8-22
- syntax errors 5-4

## T

---

- tags
  - for Javadoc 14-2
  - todo for Javadoc 14-7
- target
  - runtime configuration 7-10
- target VM options 5-6
- targets
  - Ant 6-7
  - build 5-4

- setting Ant properties 6-12
  - setting build task properties 6-17
- tearDown() 13-4
- templates
  - project 2-2
- Test Case wizard 13-5
  - tutorial 21-1
- test cases
  - See also* unit testing
  - creating 13-5
  - order of execution 13-4
  - running from a main() 13-16
  - setUp() 13-4
  - tearDown() 13-4
  - todo tags 13-6
- test fixtures
  - comparison fixture 13-10
  - custom 13-11
  - example 13-8
  - JDBC fixture 13-8
  - JNDI fixture 13-9
  - predefined 13-8
  - tutorial 22-1
- Test Hierarchy page 13-14
- test methods
  - requirements 13-6
- test runners 13-13
  - and editor 13-14
  - available in JBuilder 13-1
  - JBTestRunner 13-14
  - JUnit AwtUI 13-1
  - JUnit SwingUI 13-16
  - JUnit TextUI 13-16
  - setting 13-16
- test source directory 13-5
- Test Suite wizard 13-7
  - tutorial 21-1
- test suites 13-4
  - See also* unit testing
- TestCase class 13-1, 13-3
  - extending 13-4
  - setUp() 13-4
  - tearDown() 13-4
- testing
  - See also* unit testing
  - an EJB 13-11
  - server-side code 13-11
- TestRecorder class 13-10
- TestSuite class 13-1, 13-3
- TextUI 13-16
- threads
  - choosing for stepping 8-31
  - detecting deadlocks 8-32
  - displaying current 8-31
  - displaying top stack frame 8-31
  - keeping suspended 8-31
  - managing 8-30
  - split pane 8-30
- Threads, call stacks and data view
  - split pane 8-30
- Threads, call stacks, and data view 8-15
- todo tags 14-7
  - in test cases 13-6
- tool tips
  - debugger 8-26
  - UML diagrams 11-13
- toolbar
  - debugger 8-23
- tools
  - appletviewer B-1
  - command-line B-1
  - jar 15-5, B-1
  - java B-1
  - javac B-1
  - javadoc B-1
  - JBuilder command-line interface B-4
  - native2ascii B-1
- Trace Into settings
  - for classes with no source available 8-39
- tracing into classes
  - disabling 8-37
  - enabling 8-37
- try/catch
  - surrounding code block with 12-26
- tutorials
  - building with Ant files 18-1
  - compiling, running, and debugging 17-1
  - creating and running unit tests 21-1
  - remote debugging 19-1
  - test fixtures 22-1
  - UML 20-1

## U

---

- UML 11-1
  - and Java 11-2
  - defined 11-1
  - overview 11-1
  - tutorial 20-1
- UML browser 11-10
  - context menu 11-14
  - defined 11-10
  - navigating diagrams 11-15
  - refactoring code 11-21
  - tool tips 11-13
  - tutorial 20-1
  - viewing code 11-13
  - viewing Javadoc 11-13

- UML diagrams 11-3
    - class diagram defined 11-4
    - class diagrams 11-12
    - customizing 11-17
    - defined 11-7
    - filtering 11-18
    - Find References 11-21
    - including library references 11-18
    - including references from generated source 11-19
    - obfuscated code 11-3
    - package diagram defined 11-4
    - package diagrams 11-12
    - printing 11-20
    - refactoring code 11-21
    - saving as images 11-20
    - structure pane folders 11-7
    - tool tips 11-13
    - viewing classes and packages 11-10
    - viewing inner classes 11-12
    - visibility icons 11-9
  - UML images 11-20
    - printing diagrams 11-20
    - saving diagrams 11-20
  - UML options
    - customizing IDE 11-19
    - filtering packages and classes 11-18
    - including library references 11-18
    - including references from generated source 11-19
    - project properties 11-17
  - UML terms
    - corresponding Java terms 11-2
    - defined 11-2
  - Unicode 16-12
    - 16-bit format 16-14
    - 7-bit ASCII 16-15
    - defined 16-2
    - displaying 16-10
    - entering 16-10
  - unit testing 13-1
    - an EJB 13-7, 13-11, 13-12
    - Cactus 13-2, 13-11
    - comparison fixture 13-10
    - creating tests 13-4
    - custom fixture 13-11
    - Debug Test menu option 13-3
    - debugging tests 13-17
    - defined 13-1
    - goals 13-4
    - JBTestRunner 13-14
    - JDBC fixture 13-8
    - JNDI fixture 13-9
    - JUnit 13-1
    - JUnit SwingUI 13-16
    - JUnit Test Collector 13-3
    - JUnit TextUI 13-16
    - list of features 13-2
    - Run Test menu option 13-3
    - running tests 13-13
    - runtime configuration 13-3
    - runtime configurations 13-16
    - server-side code 13-2, 13-11
    - source directory 13-5
    - Test Case wizard 13-5
    - test discovery 13-3
    - test fixtures 13-8
    - test methods 13-6
    - Test Suite wizard 13-7
    - TestCase class 13-1, 13-3
    - TestSuite class 13-1, 13-3
    - tutorial 21-1, 22-1
    - unit testing stack trace filter 13-16
    - Update Classes After Compiling 8-65
  - updating
    - classes after compiling 8-64
    - classes after compiling (options) 8-65
  - Usenet newsgroups 1-7
  - user interfaces
    - designing JavaBean 10-4
- ## V
- 
- valid JavaBeans
    - checking for 10-21
  - values
    - examining while debugging 8-55
  - variable
    - introducing 12-25
  - variable watches 8-59
  - variables
    - finding definition of 12-7
    - finding references to 12-8
    - modifying values 8-63
  - viewing files 2-9
  - viewing project files 4-13
  - visibility icons
    - UML diagrams 11-9
  - visualizing code 11-1
    - See also* UML
  - VM
    - debugging with -classic 2-20
    - setting target VM 5-6
- ## W
- 
- watches 8-59
    - deleting 8-62
    - editing 8-61

- object watches 8-61
  - variable watches 8-59
- watching expressions 8-59
- web files
  - running 7-2
- wizards
  - Ant 6-7
  - Archive Builder 15-17
  - Comparison Fixture 13-10
  - Custom Fixture 13-11
  - External Build Task 6-16
  - Javadoc 14-8
  - JDBC Fixture 13-8
  - JNDI Fixture 13-9

- Native Executable Builder 15-29
- Resource Strings 16-5
- Test Case 13-5
- Test Suite 13-7
- working directory 4-12
- writeObject() 10-21

## Z

---

- ZIP files 15-2
  - adding to project 4-1
  - creating with Archive Builder 15-17
  - viewing 15-6

